MAY/JUNE 2017

Java™ magazine

By and for the Java community

Libraries

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /

COVER ART BY PEDRO MURTEIRA

EDITORIAL

Editor in Chief
Andrew Binstock

Managing Editor
Claire Breen

Copy Editors
Karen Perkins, Jim Donahue

Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Senior Designer
Arianna Pucherelli

Designer
Jaime Ferrand

Senior Publication Designer
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Sales Director
Tom Cometa

Account Manager
Mark Makinney

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

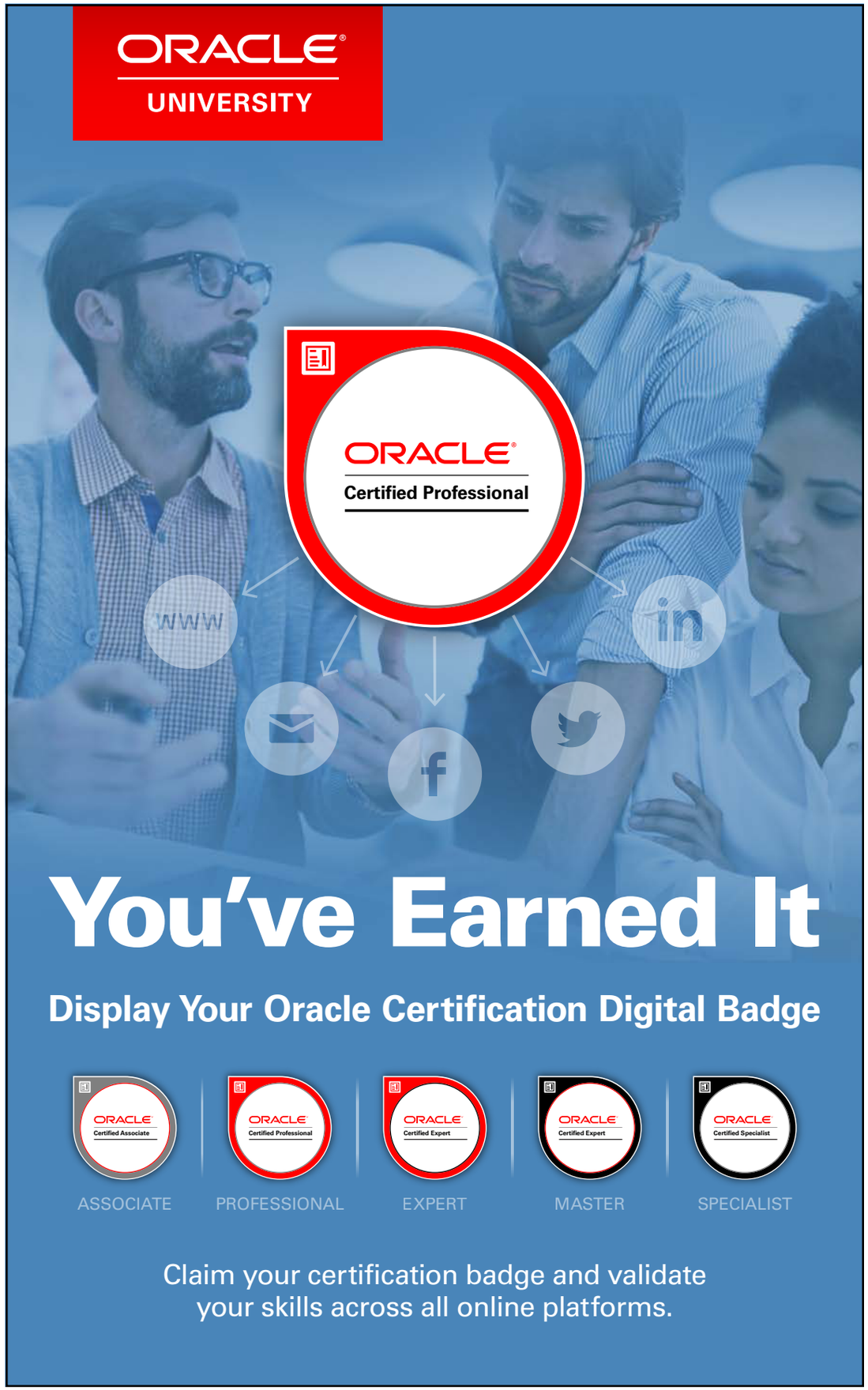
java@omeda.com

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Copyright © 2017, Oracle and/or its affiliates. All Rights Reserved. No part of this publication may be reprinted or otherwise reproduced without permission from the editors. *JAVA MAGAZINE* IS PROVIDED ON AN "AS IS" BASIS. ORACLE EXPRESSLY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS OR IMPLIED. IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY DAMAGES OF ANY KIND ARISING FROM YOUR USE OF OR RELIANCE ON ANY INFORMATION PROVIDED HEREIN. Opinions expressed by authors, editors, and interviewees—even if they are Oracle employees—do not necessarily reflect the views of Oracle. The information is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle. Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Java Magazine is published bimonthly and made available at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL–3A, Redwood City, CA 94065-1600.



Keeping Annotations Useful

When designing annotations, be conservative and circumspect.

In this issue, the lead feature is about a rightly well-regarded library named Project Lombok. This library enables you to avoid writing boilerplate code by using annotations. For example, add `@Data` to a class and Lombok will generate the getters and setters, plus other methods you're likely to want in a JavaBean-style data class—`toString()` and so on. Lombok's approach is attractive to me—and to many developers—in part because it's useful, compact, and clear. In addition, the project includes a tool called "delombok," which can remove the annotations and insert the boilerplate code into your classes. In this way, you can easily remove the dependency on Lombok. The annotations are conservative in their expression and their roles, and the project has a clean, reversible implementation. Many more annotations should follow this approach.

Annotations as they appear in Java itself follow a similar understated model. Those annotations, first unveiled in Java 5, were elegant and concise and didn't attempt to do too much: `@Override` and `@Deprecated` told you something about the code, while `@SuppressWarnings` told the tools that you knew what you were doing. The intent was that tools, especially IDEs, would use these markers to issue warnings and reminders. None of the annotations actually changed program behavior. This conservative approach by the Java language team continued in Java 7, when `@SafeVarargs` was added, and in Java 8, when `@FunctionalInterface` was delivered.

In addition to the qualities I've already mentioned, these annotations are unambiguous. This is a key and often overlooked aspect of annotation design. In the quest for brevity, annotation

PHOTOGRAPH BY BOB ADLER/THE VERBATIM AGENCY

ORACLE

Java in the Cloud

Oracle Cloud delivers high-performance and battle-tested platform and infrastructure services for the most demanding Java apps.

Oracle Cloud.
Built for modern app dev.
Built for you.

Start here:
developer.oracle.com

#developersrule

authors too often push the responsibility of intelligibility onto developers. Look no further than the lack of coordination in basic syntax. A prime offender here is the family of not-null qualifiers. `@NotNull` is such an annotation, and so is `@NonNull`. This annotation has a different meaning under FindBugs. Likewise, `@Nullable` means different things to the Checker Framework and FindBugs. This is not the way to go about things. If you author annotations, be clear and definitely avoid synonyms and homonyms.

In comparison to these short, pithy annotations, Java EE introduced the extensive use of annotations. Soon, a series of annotations replaced code, and the nature of enterprise programming thereby changed. Java EE acquired a sort of embedded syntax that straddled descriptions and commands. The advent of this style reinvigorated Java EE by making it far easier to code and by getting rid of the heaviness of its forebears.

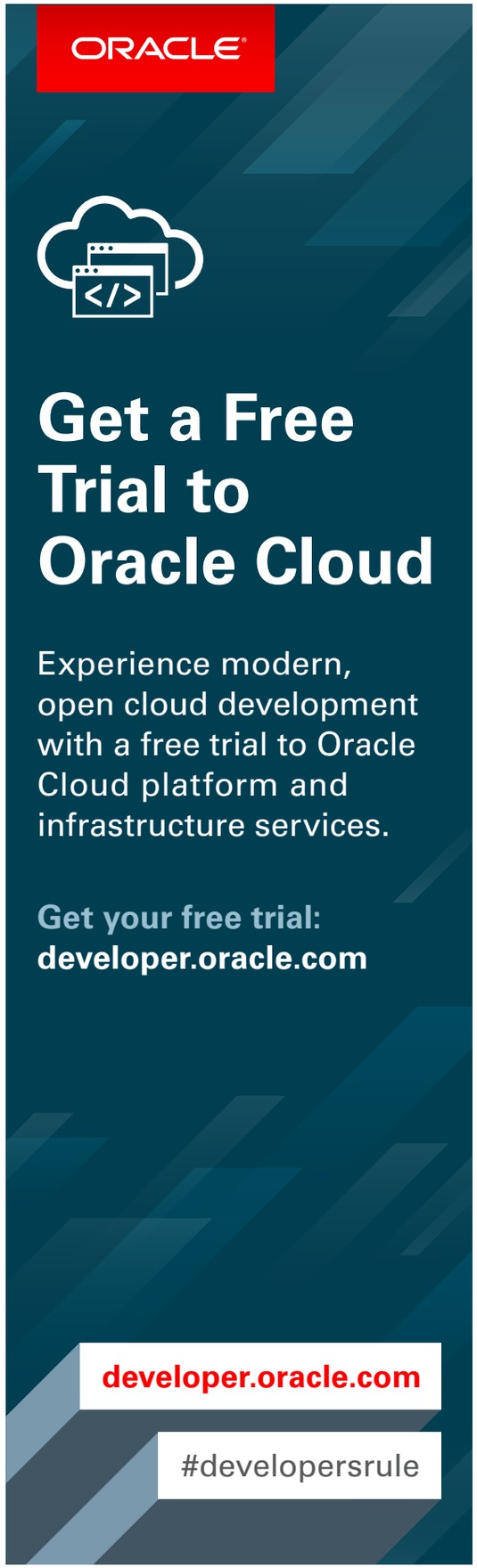
However, this advance inspired legions of frameworks to use and overuse annotations, many of which were uninspired formulations. They introduced complexity without good enough documentation by which to navigate the code. As the annotations became complex markers for actions defined elsewhere, you ended up chasing your tail just to determine what the code you had right before you actually did. This was less than entirely fun, which brings me to the second problem with many annotations: insufficient documentation. Unless the meaning is utterly transparent (and even then, as the preceding examples demonstrated), document the annotation thoroughly, especially in frameworks. Err on the side of overcommunication.

Finally, I need to stress the importance of making annotations a sound proposition for the developer and, by extension, the developer's team. In this regard, I am leery of IDE vendors' creation of their own proprietary annotation systems. All IDEs do this to some extent, but I'll pick an example from the one I use most. IntelliJ IDEA uses annotations to deliver a minimal but clever implementation of design by contract (DbC)–style enforcement of passed parameters and return values. I applaud JetBrains for providing a handy way to have the IDE enforce method contracts (and identify potential coding errors that are inconsistent with the contract requirements).

IntelliJ uses syntax like this: `@Contract("_, null -> null")`. It means that the tagged method accepts two parameters and returns a null if the second parameter is null. Much as I like this annotation, I feel uncomfortable fully committing to it because it creates a dependence on the IDE. (Even though another IDE will skip the annotation it doesn't recognize, I've now left an unused artifact in my code that might create wasted time for downstream developers trying to understand its function.) In addition, if my whole team is not using the same IDE, then some code won't have these tests and my hope of consistent DbC enforcement is either compromised or IDE-dependent.

Annotations are an important part of programming in Java, and their role is likely to expand. But new annotations should be devised with far greater circumspection than in the past, named with careful attention to predecessors, and documented well, and they should avoid the introduction of restrictive dependencies.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy



//events /

Devoxx Poland
KRAKOW, POLAND
JUNE 21–23
For three days, 100 Java Champions, evangelists, and thought leaders inspire 2,500 developers from 20 different countries at this installment of the popular Devoxx conferences. Tracks on server-side Java, cloud and big data, JVM languages, web and HTML5, and more are on offer. Hacking and networking round out the experience.

PHOTOGRAPH BY ZARNELL/GETTY IMAGES

JEEConf
MAY 26–27
KIEV, UKRAINE
JEEConf is the largest Java conference in Eastern Europe. The annual conference focuses on Java technologies for application development. This year, it offers five tracks and more than 50 speakers with an emphasis on practical experience and development of real projects. Topics include modern approaches in the development of distributed, highly loaded, scalable enterprise systems with Java, among others.

jPrime
MAY 30–31
SOFIA, BULGARIA
jPrime is a relatively new conference, with two days of talks on Java, JVM languages, mobile and web programming, and best practices. The event is run by the Bulgarian Java User Group and provides opportunities for hacking and networking.

GeekOut
JUNE 8–9
TALLINN, ESTONIA
This two-day Java developer conference focuses on Java, the JVM, programming languages and methodologies, developer tooling, solution architecture, and continuous delivery. A product exhibition is included.

JBCN Conference
JUNE 19–21
BARCELONA, SPAIN
Hosted by the Barcelona Java Users Group, this conference is dedicated to Java and JVM development. Share your knowledge and experiences, and discover how other developers are using your favorite VM.

O'Reilly Fluent Conference
JUNE 19–20, TRAINING
JUNE 20–22, TUTORIALS AND CONFERENCE
SAN JOSE, CALIFORNIA
Fluent offers practical training for building sites and apps for the modern web. This event is designed to appeal to application, web, mobile, and interactive developers, as well as engineers, architects, and UI/UX designers. Training days and tutorials round out the conference experience.

EclipseCon 2017

JUNE 20, "UNCONFERENCE"
JUNE 21–22, CONFERENCE
TOULOUSE, FRANCE

EclipseCon is all about the Eclipse ecosystem. Contributors, adopters, extenders, service providers, consumers, and business and research organizations gather to share their expertise. The two-day conference is preceded by an "Unconference" gathering.

QCon New York

JUNE 26–28, CONFERENCE
JUNE 29–30, WORKSHOPS
NEW YORK, NEW YORK

QCon is a practitioner-driven conference for technical team leads, architects, engineering directors, and project managers who influence innovation in their teams. The conference covers many different developer topics, frequently including entire Java tracks.

Java Forum

JULY 5, WORKSHOP
JULY 6, CONFERENCE
STUTTGART, GERMANY

Organized by the Stuttgart Java User Group, Java Forum typically draws more than 1,000 participants. A workshop for Java decision-makers takes place on July 5. The broader forum will be held on July 6, featuring 40 exhibitors and including lectures, presentations, demos, and Birds of a Feather sessions. (No English page available.)

The Developer's Conference (TDC)

JULY 11–15
SÃO PAULO, BRAZIL

TDC is one of Brazil's largest conferences for students, developers, and IT professionals. Java-focused content on topics such as IoT, UX design, mobile development, and functional programming are featured. (No English page available.)

JCrete

JULY 16–21
KOLYMBARI, GREECE

This loosely structured "unconference" involves morning sessions discussing all things Java, combined with afternoons spent socializing, touring, and enjoying the local scene. There is also a JCrete4Kids component for introducing youngsters to programming and Java. Attendees often bring their families.

ÜberConf

JULY 18–21
DENVER, COLORADO

ÜberConf 2017 will be held at the Westin Westminster in downtown Denver. Topics include Java 8, microservice architectures, Docker, cloud, security, Scala, Groovy, Spring, Android, iOS, NoSQL, and much more.

JavaZone 2017

SEPTEMBER 12, WORKSHOPS
SEPTEMBER 13–14, CONFERENCE
OSLO, NORWAY

JavaZone is a conference for Java developers created by the Norwegian Java User Group, javaBin. The conference has existed since 2001 and now consists of around 200 speakers and 7 parallel tracks over 2 days, plus an additional day of workshops beforehand. You will be joined by approximately 3,000 of your fellow Java developers. Included in the ticket price is a membership in javaBin.

NFJS Boston

SEPTEMBER 29–OCTOBER 1
BOSTON, MASSACHUSETTS

Since 2001, the No Fluff Just Stuff (NFJS) Software Symposium Tour has delivered more than 450 events with more than 70,000 attendees. This event in Boston covers the latest trends within the Java and JVM ecosystem, DevOps, and agile development environments.

PHOTOGRAPH BY CHRISTIAN REIMER/FLICKR

JavaOne

OCTOBER 1–5
SAN FRANCISCO, CALIFORNIA
Whether you are a seasoned coder or a new Java programmer, JavaOne is the ultimate source of technical information and learning about Java. For five days, Java developers gather from around the world to talk about upcoming releases of Java SE, Java EE, and JavaFX; JVM languages; new development tools; insights into recent trends in programming; and tutorials on numerous related Java and JVM topics.

KotlinConf

NOVEMBER 2–3
SAN FRANCISCO, CALIFORNIA
KotlinConf is a JetBrains event that provides two days of content from Kotlin creators and community members.

Devoxx

NOVEMBER 6–10
ANTWERP, BELGIUM
The largest gathering of Java developers in Europe takes place again this year in Antwerp. Dozens of expert speakers deliver hundreds of presentations on Java and the JVM. Tracks include server-side Java, cloud, big data, and extensive coverage of Java 9.

QCon San Francisco

NOVEMBER 13–15, CONFERENCE
NOVEMBER 16–17, WORKSHOPS
SAN FRANCISCO, CALIFORNIA
Although the content has not yet been announced, recent QCon conferences have offered several Java tracks along with tracks related to web development, DevOps, cloud computing, and more.

Are you hosting an upcoming Java conference that you would like to see included in this calendar? Please send us a link and a description of your event at least 90 days in advance at javamag_us@oracle.com. Other ways to reach us appear on the last page of this issue.

Oracle Code Events

Oracle Code is a free event for developers to learn about the latest development technologies, practices, and trends, including containers, microservices and API applications, DevOps, databases, open source, development tools and low-code platforms, machine learning, AI, and chatbots. In addition, Oracle Code includes educational sessions for developing software in Java, Node.js, and other programming languages and frameworks using Oracle Database, MySQL, and NoSQL databases.

US AND CANADA

JUNE 22, Atlanta, Georgia

EUROPE AND MIDDLE EAST

MAY 23, Moscow, Russia
JUNE 6, Brussels, Belgium
JULY 11, Tel Aviv, Israel

ASIA PACIFIC

JULY 14, Beijing, China
JULY 18, Sydney, Australia
AUGUST 4, Bangalore, India
AUGUST 30, Seoul, South Korea

LATIN AMERICA

JUNE 27, São Paulo, Brazil
JUNE 29, Mexico City, Mexico

Oracle Press™

Your Destination for Java Expertise

Written by leading Java experts, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.

Raspberry Pi with Java: Programming the Internet of Things (IoT)

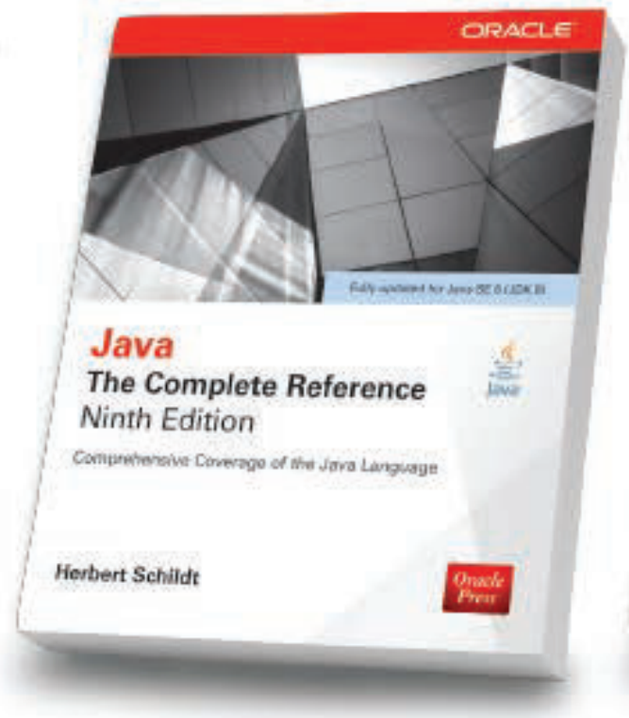
Stephen Chin, James Weaver

Use Raspberry Pi with Java to create innovative devices that power the internet of things.

Introducing JavaFX 8 Programming

Herbert Schildt

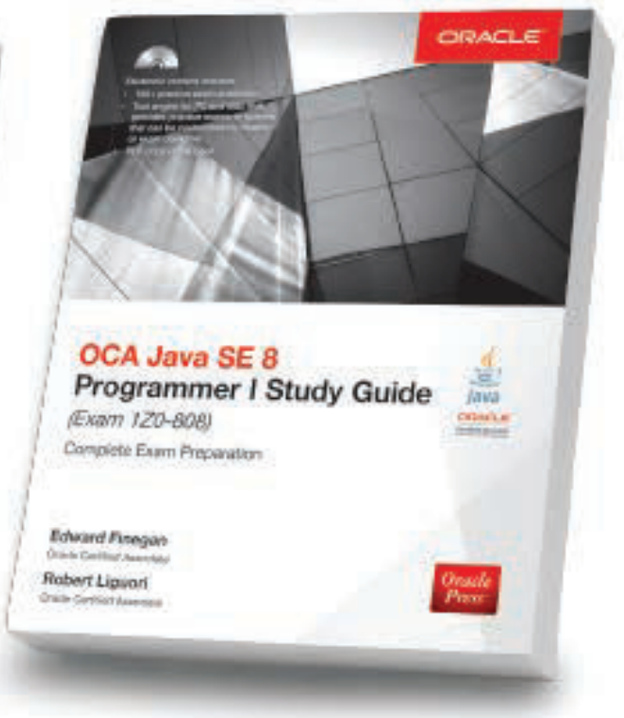
Learn how to develop dynamic JavaFX GUI applications quickly and easily.



Java: The Complete Reference, Ninth Edition

Herbert Schildt

Fully updated for Java SE 8, this definitive guide explains how to develop, compile, debug, and run Java programs.



OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808)

Edward Finegan, Robert Liguori

Get complete coverage of all objectives for Exam 1Z0-808. Electronic practice exam questions are included.

Available in print and as eBooks

Oracle Press™

www.OraclePressBooks.com • @OraclePress

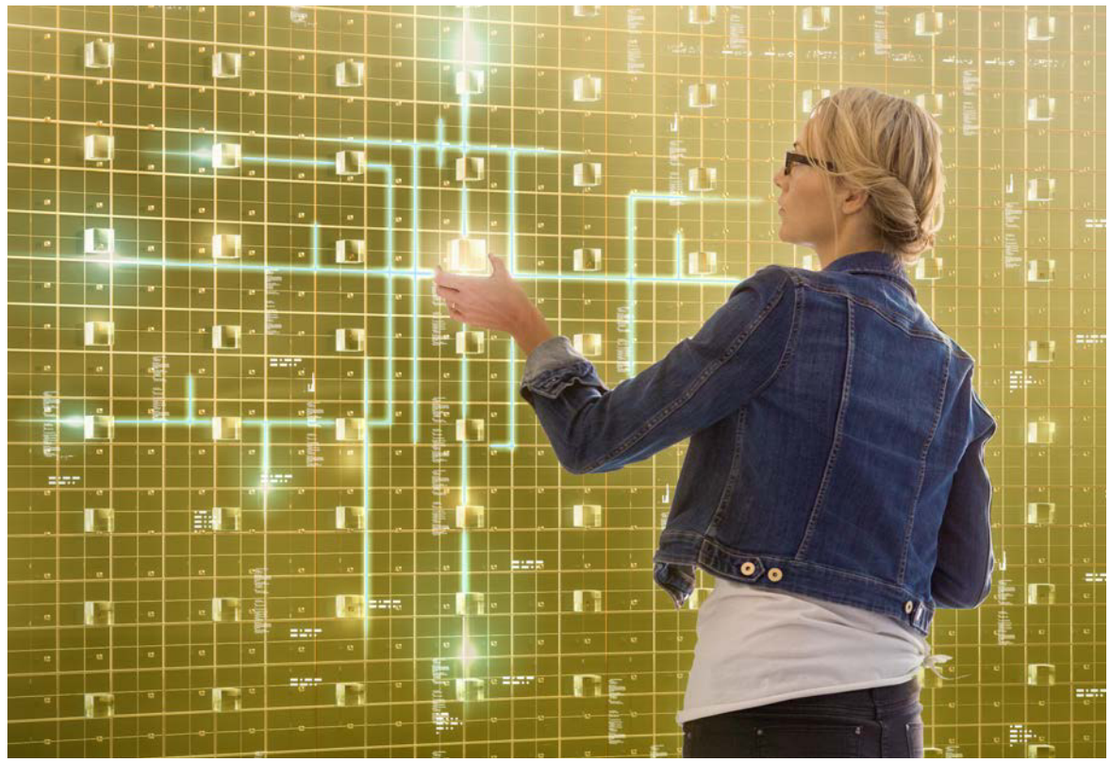
Writing and Using Libraries

In an age of frameworks, there still remains a supreme need for libraries, those useful collections of classes and methods that save us a huge amount of work. For all the words spilled on the reusability of object orientation (OO), it's clear that code reuse has been consistently successful only at the library level. It's hard to say whether that's a failure of the promises of OO or whether those promises were unlikely to ever deliver the hoped-for reusability.

In Stephen Colebourne's article (page 28), he gives best practices for writing libraries of your own. Colebourne is the author of the celebrated Joda-Time library, which was the standard non-JDK time and date library prior to Java SE 8. In the article, he gives best practices for architecting the library and shares guidelines he has learned along the way that sometimes fly in the face of generally accepted programming precepts. Writing your own library? Then start here.

We also examine three well-designed libraries that provide useful functionality but might not be widely known. The first of these is Project Lombok (page 10), which uses annotations to greatly reduce the writing of boilerplate code—leading to fewer keystrokes and much more readable code.

Andrés Almiray's article on the JDeferred library (page 16) is a deep dive into the concepts of futures and promises, which are techniques for defining, invoking, and getting results from asynchronous operations. The built-in Java classes for futures and promises work well but can be difficult to program. JDeferred removes the difficulty and, like Lombok, leads to considerably cleaner code.



Finally, we revisit an article we ran a year ago on jsoup (page 22), which is one of the finest ways of handling HTML: parsing, scraping, manipulating, and even generating it.

If libraries are not your favorite topic, we have you covered with a detailed discussion (page 34) of how to use streaming syntax rather than SQL when accessing databases. In addition, we offer our usual quiz (this time with the inclusion of questions from the entry-level exam), our calendar of events, and other goodness. (Note that our next issue will be a jumbo special issue on Java 9.) Enjoy!

ART BY PEDRO MURTEIRA

JOSH JUNEAU

Project Lombok: Clean, Concise Code

Add Lombok to your project and get rid of most of your boilerplate code.

Imagine that you are coding a Java application and creating a plain old Java object (POJO), a Java class with several private fields that will require getter and setter methods to provide access. How many lines of code will be needed to generate getters and setters for each of the fields? Moreover, adding a constructor and a `toString()` method will cause even more lines of code and clutter. That is a lot of boilerplate code. How about when you are utilizing Java objects that need to be closed after use, so you need to code a `finally` block or use try-with-resources to ensure that the object closing occurs? Adding `finally` block boilerplate to close objects can add a significant amount of clutter to your code.

Project Lombok is a mature library that reduces boilerplate code. The cases mentioned above cover just a few of those where Project Lombok can be a great benefit. The library replaces boilerplate code with easy-to-use annotations. In this article, I examine several useful features that Project Lombok provides—making code easier to read and less error-prone and making developers more productive. Best of all, the library works well with popular IDEs and provides a utility to "delombok" your code by reverting—that is, adding back all the boilerplate that was removed when the annotations were added.

Check for Nulls

Let's start with one of the most basic utilities that Lombok has to offer. The `@NonNull` annotation, which should not be confused with the Bean Validation annotation, can be used to generate a null check on a setter field. The check throws a `NullPointerException` if the annotated class field contains a null value. Simply apply it to a field to enforce the rule:

```
@NonNull @Setter
private String employeeId;
```

This code generates the following code:

```
public id setEmployeeId(@NonNull final String employeeId)
{
    if(employeeId == null) throw
        new java.lang.NullPointerException("employeeId");
    this.employeeId = employeeId;
```

Primitive parameters cannot be annotated with `@NonNull`. If they are, a warning is issued and no null check is generated.

Concise Data Objects

Writing a POJO can be laborious, especially if there are many fields. If you are developing a POJO, you should always provide private access directly to the class fields, while creating accessor methods—getters and setters—to read from and write to those fields. Although developing accessor methods is easy, they generally are just boilerplate code. Lombok can

take care of generating these methods if a field is annotated with `@Getter` and `@Setter`. Therefore, the following two code listings provide the exact same functionality.

Without Project Lombok:

```
private String columnName;

public String getColumnName(){
    return this.columnName;
}
public void setColumnName(String columnName){
    this.columnName = columnName;
}
```

Using Project Lombok:

```
@Getter @Setter private String columnName;
```

As you can see, Lombok not only makes the code more concise, but it also makes the code easier to read and less error-prone. These annotations also accept an optional parameter to designate the access level if needed. More good news: `@Getter` and `@Setter` respect the proper naming conventions, so generated code for a Boolean field results in accessor methods beginning with *is* rather than *get*. If they are applied at the class level, getters and setters are generated for each nonstatic field within the class.

In many cases, data objects also should contain the `equals()`, `hashCode()`, and `toString()` methods. This boilerplate can be taken care of by annotating a class with the `@EqualsAndHashCode` and `@ToString` annotations, respectively. These annotations cause Lombok to generate the respective methods, and they are customizable so that you can specify field exclusions and other factors. By default, any nonstatic or nontransient fields are included in the logic that is used to compose these methods. These annotations use the attribute `exclude` to specify methods that should not be included in the logic. The `callSuper` attribute accepts a `true` or `false`, and it indicates whether to use the `equals()` method of the superclass to verify equality. The following code demonstrates the use of these annotations.

```
@EqualsAndHashCode
@ToString(exclude={"columnLabel"})
public class ColumnBean {
    private BigDecimal id;
    private String columnName;
    private String columnLabel;
}
```

Lombok can take care of the logger declaration if you place the @Log annotation (or an annotation pertaining to your choice of logging API) on any class that requires logging capability.

The `@Data` annotation can be used to apply functionality behind all the annotations discussed thus far in this section. That is, simply annotating a class with `@Data` causes Lombok to generate getters and setters for each of the nonstatic class fields and a class constructor, as well as the `toString()`, `equals()`, and `hashCode()` methods. It also creates a constructor that accepts any final fields or those annotated with `@NonNull` as arguments. Finally, it generates default `toString()`, `equals()`, and `hashCode()` methods that take all class fields and methods into consideration. This makes the coding of a POJO very easy, and it is much the same as some alternative languages, such as Groovy, that offer similar features. **Listing 1** (all listings for this article can be found in *Java Magazine*'s download section) shows the full Java code for the POJO that is generated by the following code:

```
@Data
public class ColumnBean {
    @NonNull
    private BigDecimal id;
    @NonNull
    private String columnName;
    @NonNull
    private String columnLabel;
}
```

Note that if you create your own getters or setters, Lombok does not generate the code even if the annotations are present. This can be handy if you wish to develop a custom getter or setter for one or more of the class fields.

If you are merely interested in having constructors generated automatically, `@AllArgsConstructor` and `@NoArgsConstructor` might be of use. `@AllArgsConstructor` creates a constructor for the class using all the fields that have been declared. If a field is added or removed from the class, the generated constructor is revised to accommodate this change. This behavior can be convenient for ensuring that a class constructor always accepts values for each of the class fields. The disadvantage of using this annotation is that reordering the class fields causes the constructor arguments to be reordered as well, which could introduce bugs if there is code that depends upon the position of arguments when generating the object. `@NoArgsConstructor` simply generates a no-argument constructor.

The `@Value` annotation is similar to the `@Data` annotation, but it generates an immutable class. The annotation is placed at the class level, and it invokes the automatic generation of getters for all private final fields. No setters are generated, and the class is marked as final. Lastly, the `toString()`, `equals()`, and `hashCode()` methods are generated, and a constructor is generated that contains arguments for each of the fields.

Can't My IDE Do That?

You might be asking yourself, "Can't my IDE already do that sort of refactoring?" Most modern IDEs—such as NetBeans, Eclipse, and IntelliJ—offer features such as encapsulation of fields and auto-generation of code. These abilities are great because they can significantly increase productivity. However, these capabilities do not reduce code clutter, so they can lead to refactoring down the road. Let's say your Java object has 10 fields. To conform to a JavaBean, it will contain 20 accessor methods (one getter and setter pair per field). That's a lot of clutter. Also, what happens when you decide to change one of your field names? You'll have to do some refactoring in order to change it cleanly. If you're using Lombok, you simply change the field name and move on with your life.

Builder Objects

Sometimes it is useful to have the ability to develop a builder object, which allows objects to be constructed using a step-by-step pattern with controlled construction. For example, in some cases large objects require several fields to be populated, which can be problematic when such an object is implemented via a constructor.

Lombok makes it simple to create builder objects in much the same way that it enables easy POJO creation. Annotating a class with `@Builder` produces a class that adheres to the builder pattern—that is, an inner builder class is produced that contains each of the class fields. ("Builder" is preceded by the name of the class. So a class named `Foo` has a `FooBuilder` class generated.) The generated builder class contains a "setter" method for each of the class fields, but the names of the methods do not include the usual "set" prefix. The methods themselves set the value that is passed into the methods, and then they return the builder object. **Listing 2** in the downloadable code demonstrates a class that contains a builder, and **Listing 3** demonstrates the same object annotated with `@Builder`.

Several variations can be used with `@Builder`. For example, the annotation can be placed on the class, on a constructor, or on a method. Placing the annotation on a constructor produces the same builder object shown in **Listing 2**, but it generates methods for each of the constructor's arguments in the builder. This means that you can omit a class field from the constructor, or you can choose to include a superclass field in the constructor. The only way to include superclass fields in a builder is for an object to contain a superclass.

The `toBuilder` attribute of the `@Builder` annotation accepts `true` or `false`, and it can be used to designate whether a `toBuilder()` method is included in the generated builder object. This method copies the contents of an existing object of the same type.

It is possible to treat one of the fields as a builder collection by annotating it with `@Singular`. This causes two adder methods to be generated—one to add a single element and another to add all elements. This annotation also causes a `clear()` method to be generated, which clears the contents of the collection.

Easy Cleanup

Lombok makes it easy to clean up resources as well. How often have you either forgotten to close a resource or written lots of boilerplate `try-catch` blocks to accommodate resource closing? Thanks to the `@Cleanup` annotation, you no longer need to worry about forgetting to release a resource.

Although the Java language now contains the try-with-resources statement to help close resources, `@Cleanup` can be a useful alternative in some cases, because it causes a `try-finally` block to be generated around the subsequent code, and then it calls the annotated resource's `close()` method. If the cleanup method for a given resource is not named `close()`, the cleanup method name can be specified with the annotation's `value` attribute. **Listing 4** in the downloadable code demonstrates a block of code that contains some lines to manually close the resource. **Listing 5** demonstrates the same block of code using `@Cleanup`.

It is important to note that in a case where code throws an exception and then subsequent code invoked via `@Cleanup` also throws an exception, the original exception will be hidden by the subsequently thrown exception.

Locking Safely

To ensure safety by having only one thread that can access a specified method at a time, the method should be marked as `synchronized`. Lombok supplies an even safer way to ensure that only one thread can access a method at a time: the `@Synchronized` annotation. This annotation can be used only on static and instance methods, just like the `synchronized` keyword. However, rather than locking on `this`, the annotation locks on a private field named `$lock` for nonstatic methods and on `$LOCK` for static methods. This field is autogenerated if it does not already exist, or you can create it yourself. You can also specify a different lock field by specifying it as a parameter to `@Synchronized`. The following code illustrates the use of `@Synchronized`:

```
@Synchronized
public static void helloLombok(){
    System.out.println("Lombok");
}
```

This solution can be a safer alternative to using the `synchronized` keyword, because it allows you to lock on an instance field rather than on `this`.

Effortless Logging

Most logging requires some declaration to set up a logger within each class. This code is definitely repetitive boilerplate code. Lombok can take care of the logger declaration if you place the `@Log` annotation (or an annotation pertain-

ing to your choice of logging API) on any class that requires logging capability.

For instance, if you wish to use a logging API—say, Log4j 2—each class that uses the logger must declare something similar to the following:

```
public class ClassName(){
private static final org.apache.log4j.Logger log =
    org.apache.log4j.Logger.getLogger(ClassName.class);
. . .
// Use log variable as needed
}
```

Lombok makes it possible to do the following instead:

```
@Log4j2
public class ClassName(){
. . .
// Use log variable as needed
}
```

Listing 6 in the downloadable code shows an example using Log4j 2. The name of the logger will automatically be the same as its containing class' name. However, this can be customized by specifying the topic attribute of the respective logging annotation. For a complete listing of supported logging APIs, refer to the Lombok documentation and the Lombok Javadoc.

Other Useful Items

There are several other useful features Lombok offers that I haven't yet covered. Let's go through a couple of the most highly used.

Informal declaration. The `val` keyword can be used in place of an object type when you declare a local final variable, much like the `val` keyword that you have seen in alternative languages such as Groovy or Jython. Take the following code, for instance:

```
final ArrayList<Job> myJobs = new ArrayList<Job>();
```

Using the `val` keyword, you can change the code to the following:

```
val myJobs = new ArrayList<Job>();
```

There are some considerations for using the `val` keyword. First, as mentioned previously, it marks the method declaration as final. Therefore, if you later need to change the value of the variable, using the `val` keyword is not possible. It also does not work correctly in some IDEs, so if you are trying to mark local variables as final in those IDEs, they are flagged as errors.

Be sneaky with exceptions. There are occasions where exception handling can become a burden, and I'd argue that this is typically the case when you are working with boilerplate exceptions. Most of the time, Java allows you to easily see where problems exist via the use of checked exceptions. However, in those cases where checked exceptions are burdensome, you can easily hide them using Lombok.

The `@SneakyThrows` annotation can be placed on a method to essentially "swallow" the exceptions, allowing you to omit the `try-catch` block completely. The annotation allows a method to handle all exceptions quietly, or you can specify exactly which exceptions to ignore by passing the exception classes to the annotation as attributes. **Listing 7** in the downloadable code demonstrates the use of `@SneakyThrows` specifying which exceptions to swallow.

The delombok utility can be applied to your code to convert code that uses Lombok back to vanilla Java.

I want to reiterate that this Lombok feature should be used with caution, because it can become a real issue if too many exceptions are ignored.

Lazy getters. It is possible to indicate that a field should have a getter created once, and then the result should be cached for subsequent invocations. This can be useful if your getter method is expensive as far as performance goes. For instance, if you need to populate a list from a database query, or you need to access a web service to obtain the data for your field on the first access, it might make sense to cache the result for subsequent calls. To use this feature, a private final variable must be generated and initialized with the expensive expression. You can then annotate the field with `@Getter(lazy=true)` to implement this functionality.

IDE compatibility. Lombok plays well with the major IDEs, so simply including Lombok in your project and annotating accordingly typically does not generate errors in code or cause errors when the generated methods are called. In fact, in NetBeans the class `Navigator` is populated with the generated methods after annotations are placed and the code is saved, even though the methods do not appear in the code. Auto-completion works just as if the methods were typed into the class, even when generated properties are accessed from a web view in expression language.

Even more-concise Java EE. Over the past few years, Java EE has been making good headway on becoming a very productive and concise platform. Those of you who recall the laborious J2EE platform can certainly attest to the great number of improvements that have been made. I was very happy to learn that Lombok plays nicely with some Java EE APIs, such as Java Persistence API (JPA). This means it is very easy to develop constructs such as entity classes without writing all the boilerplate, which makes the classes much more concise and less error-prone. I've developed entire Java EE applications without any getters or setters in my entity classes, just by annotating them with `@Data`. I suggest you play around with it and see what works best for you.

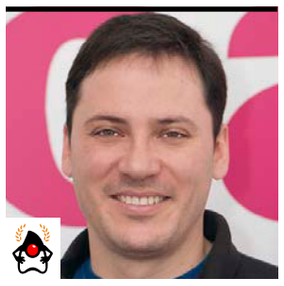
Use caution and roll back. As with the use of any library, there are some caveats to keep in mind. This is especially true when you are thinking about future maintenance or modifications to the codebase. Lombok generates code for you, but that might cause a problem when it comes to refactoring. It is difficult to refactor code that does not exist until compile time, so be cautious with refactoring code that uses Lombok. You also need to think about readability. Lombok annotations might make troubleshooting a mystery for someone who is not familiar with the library—and even for those who are—if something such as `@SneakyThrows` is hiding an exception.

Fortunately, Lombok makes it easy to roll back if you need to. The delombok utility can be applied to your code to convert code that uses Lombok back to vanilla Java. This utility can be used via Ant or the command line.

Conclusion

The Lombok library was created to make Java an easier language in which to code. It takes some of the most common boilerplate headaches out of the developer's task list. It can be useful for making your code more concise, reducing the chance for bugs, and speeding up development time. Try adding Lombok to one of your applications and see how many lines of code you can cut out. `</article>`

Josh Juneau (@javajuneau) is a Java Champion and a member of the NetBeans Dream Team. He works as an application developer, system analyst, and database administrator. He is a frequent contributor to Oracle Technology Network and *Java Magazine*. Juneau has written several books on Java and Java EE for Apress, and he is a JCP Expert Group member for JSR 372 (JavaServer Faces [JSF] 2.3) and JSR 378 (Portlet 3.0 Bridge for JSF 2.2).



ANDRÉS ALMIRAY

JDeferred: Simple Handling of Promises and Futures

Asynchronous operations without the headaches

Developers are quite capable of dealing with events that occur serially. However, we struggle with parallel and delayed or deferred events. Fortunately, there are techniques that can help to deal with delayed or deferred results. Principal among these techniques are *promises* and *futures*, which are the focus of this article, along with a library, JDeferred, that greatly simplifies their use.

Wikipedia defines the key concept behind them as an object that acts as a proxy for a result that's initially unknown. A *future* is a read-only placeholder view of a variable; that is, its role is to contain a value and nothing more. A *promise* is a writable, single-assignment container that sets the value of the future. Promises may define an API that can be used to react to a future's state changes, such as the value being resolved, the value being rejected due to an error (expected or unexpected), or the cancelation of the computing task. Let's look at this in more detail.

Promises in Java

The standard Java library includes various implementations of the future concept based on `java.util.concurrent.Future<V>`, with one recent addition made in Java 8 named `CompletableFuture`. This class delivers the following abilities:

- Obtain a value that might be calculated in an asynchronous fashion.
- Register mutator functions that affect the calculated result, when it is ready.
- Establish a chain of functions that accept the result, potentially combining it with other results.
- Initialize a background task that computes the expected result.

You can get started quickly with `CompletableFuture` (I refer to this type as a promise from now on) by using a pair of factory methods found in this type. You can create a promise that returns no value by invoking the following:

```
CompletableFuture.runAsync(new Runnable() { ... });
```

This version allows you to define a task that performs some computation, but the result is not important. What's important is whether the task was successfully completed or not. You can attach a reaction, such as the following:

```
CompletableFuture<Void> promise =
  CompletableFuture.runAsync(() -> ...);
promise.thenApply(result -> {
  System.out.println("Task is finished!");
});
```

If you're interested in the computed result, you must invoke a

different factory method, one that takes a Supplier as argument, such as this:

```
CompletableFuture<String> promise =
  CompletableFuture.runAsync(() -> "hello");
promise.thenApply(result -> {
  System.out.println("Task result was " + result);
});
```

Once you have a reference to a promise, you can decorate it with further operations that can react to the result being computed, to an exception being thrown during computation, or to additional transformations to the returned value.

Now let's say that you've been asked to display a list of repositories using the name of an organization found on GitHub. This requires you to invoke a REST API call, process the results, and display them. Let's further assume that the code must be assembled as a JavaFX application. This last requirement forces you to think about using the concept of a promise, because the computation of the repository list must be executed in a background thread, but the result must be published inside the UI thread—that's the general rule when building interactive JavaFX applications. Stated otherwise, any operation that's not related to the UI (such as a network call, in our case) must occur in a thread that's not the UI thread; conversely, any operation that's UI related (such as updating a widget's properties) must occur inside the UI thread. I won't get into the details of how the actual network call is produced; however, the full working code can be found on GitHub. The following snippet shows how to run the computation in the background using a promise. In this project, you'll see that I inject some of the related resources:

```
public class GithubImpl implements Github {
  @Inject private GithubAPI api;
  @Inject private ExecutorService executorService;

  @Override
  public
  CompletableFuture<List<Repository>> repositories(
      final String organization) {
    Supplier<List<Repository>> supplier = () -> {
      Response<List<Repository>> r = null;
      try {
        r = api.repositories(organization).execute();
      } catch (IOException e) {
        throw new IllegalStateException(e);
      }

      if (r.isSuccessful()) {
        return r.body();
      }
      throw new IllegalStateException(r.message());
    };

    return CompletableFuture.supplyAsync(supplier,
            executorService);
  }
}
```

The code shows the network call being issued, using `execute()`. If a communication problem or a parsing error occurs, an `IOException` is thrown. If the call was successful, the parsed body is returned; if it was not successful, an `IllegalStateException` is thrown. Finally, the promise is created by specifying a target `Executor`. You might notice in the previous snippet that I

Pay close attention to the order of the steps used to process the result supplied by the promise. If the steps are sequenced in a different order, you'll end up with different, and perhaps unexpected, behavior.

did not define an explicit executor. This is because the common `ForkJoin` pool is used if no `Executor` is defined.

Now, let's consume the promised result. I'll assume there's another component (a controller, for example) whose responsibility is to invoke the service that was just defined and populate a list with the results. It also has the responsibility to display an error if an exception occurs during the invocation of the service.

JDeferred allows you to group callbacks by responsibility, thereby eliminating the ordering problem with CompletableFuture.

```
public class AppController {
  @Inject private AppModel model;
  @Inject private Github github;
  @Inject private ApplicationEventBus eventBus;

  public void loadRepositories() {
    model.setState(RUNNING);
    github.repositories(model.getOrganization())
      .thenAccept(model.getRepositories()::addAll)
      .exceptionally(t -> {
        eventBus.publishAsync(new ThrowableEvent(t));
        return null;
      })
      .thenAccept(result -> model.setState(READY));
  }
}
```

I'll shortly explain how the last line is executed. Let's deconstruct the code snippet above line by line. First, the controller sets some state, which is used by the UI to disable further actions until the computation is finished. Next, it invokes the service and obtains a promise, `repositories`, described in the previous snippet. The promise allows the controller to set further actions, such as processing the result—in this case, adding the list of repositories to a model that is likely used by the UI for display. It then handles any possible exceptions that might have occurred during the execution of the service, using the lambda in `exceptionally()`. Finally, it sets the state again, regardless of success or failure, with the lambda in `thenAccept()`.

Caveats with Expectations

Pay close attention to the order of the steps used to process the result supplied by the promise. If the steps are sequenced in a different order, you'll end up with different, and perhaps unexpected, behavior. Let's label the steps as SUCCESS, FAILURE, ALWAYS. The current working order is thus: SUCCESS, FAILURE, ALWAYS.

If you use a different sequence, it will produce different results:

- ALWAYS, SUCCESS, FAILURE will not even compile, because the ALWAYS changes the result type to `Void` as a stand-in for the return from the lambda when a value is not returned.
- SUCCESS, ALWAYS, FAILURE causes the UI to remain disabled if an error occurred, because the model state the UI is waiting on is never updated.
- FAILURE, SUCCESS, ALWAYS also causes the UI to remain disabled if an error occurred—again, because the state is not updated.

So, you must be very vigilant regarding the order of actions attached to this type of promise. There's another inherent problem in `CompletableFuture`: the fact that it is both a future and a promise. Promises allow you to react in an asynchronous fashion that is nonblocking. However, `Future` has one particular method that is blocking in nature: `get()`. This means you can turn a nonblocking scenario into a blocking one at any time—even inadvertently, because it's so common

to call `get()` on types that expose such a method (for example, `Optional`).

You might ask, "What's the big deal? As long as I don't call the get method, everything should be fine, right?" But given that this type of promise is a future, there's no guarantee its get method won't be called further down the stream by another API that can handle futures. It would be much better if this promise were not a future in the first place. The next question might be, "What if I wrap `CompletableFuture` with a promise-only API?" Yes, that would work, but what about switching to a ready-made promise library? I'm talking about JDeferred.

Introducing JDeferred

JDeferred is a library that delivers the concept of promises. It is inspired by JQuery and Android Deferred Object. It is designed to be compatible with JDK 1.6 and later. Its API is very simple, but don't be fooled by this simplicity—you can build stable, well-behaved, readable code with it. Let's revisit the previous example using JDeferred. The full code is available on GitHub, if you want to study it in detail.

JDeferred can be added to your project with the following Maven entries:

```
<dependency>
    <groupId>org.jdeferred</groupId>
    <artifactId>jdeferred-core</artifactId>
    <version>1.2.5</version>
</dependency>
```

Or if you prefer Gradle, use this:

```
compile 'org.jdeferred:jdeferred-core:1.2.5'
```

JDeferred offers a basic type, `org.jdeferred.Promise`, that can be used to register actions or callbacks. A `Promise` may return a value upon completion, throw an `Object` (any `Object`—not just `Throwable`) if an error occurs, and return intermediate results during computation. The last two options are not possible with `CompletableFuture`. JDeferred allows you to group callbacks by responsibility, thereby eliminating the ordering problem discussed earlier with `CompletableFuture`. Promises are usually created by another component called the `DeferredManager`. In this way, the library decouples the task-creation mechanism from the promise itself, because these are two distinct concepts. Let's see how the implementation of the previous GitHub service with JDeferred looks now.

```
public class GithubImpl implements Github {
  @Inject private GithubAPI api;
  @Inject private DeferredManager deferredManager;

  @Override
  public Promise<List<Repository>, Throwable, Void>
      repositories(final String organization) {
    return deferredManager.when(() -> {
      Response<List<Repository>> r =
        api.repositories(organization).execute();
      if (r.isSuccessful())
        { return r.body(); }
      throw new IllegalStateException(r.message());
    });
  }
}
```

The code above is functionally equivalent to the code examined earlier, but it is considerably cleaner. Tasks executed in this way benefit from automatic error handling performed by `DeferredManager`. This is why you don't need to explicitly handle communication and parsing errors like you did before. These errors set the promise state to failed, and they are

recorded in such a way that the failure callbacks will receive them.

This example does not produce any intermediate result, which is why the third argument to `Promise` is set to `Void`.

Now, consuming the promise can be done in the following way:

JDeferred implements a simpler API that delivers the same capabilities without the drawbacks.

```
public class AppController {
  @Inject private AppModel model;
  @Inject private Github github;
  @Inject private ApplicationEventBus eventBus;

  public void loadRepositories() {
    model.setState(RUNNING);
    github.repositories(model.getOrganization())
        .done(model.getRepositories()::addAll)
        .fail(t ->
          eventBus.publishAsync(new ThrowableEvent(t)))
        .always((state, resolved, rejected) ->
          model.setState(READY));
  }
}
```

The controller performs the same functions as before, but the code is considerably cleaner. You can define the SUCCESS, FAILURE, ALWAYS callbacks in any order you deem necessary for this particular case. Finally, there's no way to force the promise to wait in a blocking manner for the result to be delivered; the API simply won't allow it.

If you want, you can also switch to a more manual implementation for producing the promise, using `DeferredObject`. This type allows you to set the computed or rejected value, as well as publish intermediate results if needed. If you've ever used the `SwingWorker` API, then you know how this behavior plays out—the key difference being that notifications sent by `DeferredObject` are sent in the background thread whereas `SwingWorker` sends them inside the UI thread. Here's how `DeferredObject` can be used to manually set a promised result or trigger a failure:

```
public class GithubImpl implements Github {
  @Inject private GithubAPI api;
  @Inject private ExecutorService executorService;

  @Override
  public Promise<List<Repository>, Throwable, Void>
  repositories(final String organization) {
    Deferred<List<Repository>, Throwable, Void> d =
      new DeferredObject<>();
    executorService.submit(() -> {
      Response<List<Repository>> r = null;
      try {
        r = api.repositories(organization).execute();
      } catch (IOException e) {
        d.reject(e);
        return;
      }
      if (r.isSuccessful()) { d.resolve(r.body()); }
      d.reject(new IllegalStateException(r.message()));
    });
    return d.promise();
  }
}
```

This time, you must handle any communication and parsing errors, as well as explicitly schedule the background task using an `Executor` or similar means. This particular usage of `DeferredObject` comes in handy when writing tests, because you can resolve or reject a promise at any time. The following test case shows exactly how such a scenario (that is, writing

tests) can be implemented using a combination of JDeferred, Mockito, and dependency injection:

```
@RunWith(JukitoRunner.class)
public class AppControllerTest {
  @Inject private AppController controller;
  @Inject private AppModel model;

  @Test
  public void happyPath(Github github) {
    // given:
    Collection<Repository> rs =
        TestHelper.createSampleRepositories();
    Promise<List<Repository>, Throwable, Void> p =
        new DeferredObject<List<Repository>,
              Throwable, Void>().resolve(rs);
    when(github.repositories("foo")).thenReturn(p);

    // when:
    model.setOrganization("foo");
    controller.loadRepositories();

    // then:
    assertThat(model.getRepositories(), hasSize(3));
    assertThat(model.getRepositories(), equalTo(rs));
    verify(github, only()).repositories("foo");
  }
}
```

Here we can see how `DeferredObject` is used to set up an expected result alongside a mocked instance of the `Github` class. This particular test checks the happy path in which everything works as expected. You could set up a failing path by invoking `rejected()` instead, checking that the expected exception occurred.

Conclusion

Promises enable you to handle computed results in a deferred or asynchronous manner. Java 8 provides a type named `CompletableFuture` that can be used as a promise. It allows handling of results; transforming results into further values; combining a result with other results; and handling exceptional cases when errors occur.

However, you must pay attention to the order in which actions are attached to such a promise. Also, it's possible to block such a promise at any time by simply invoking the `get()` method. JDeferred implements a simpler API that delivers the same capabilities without the drawbacks. It also allows you to publish intermediate results at any time during the background computation. Examples of this latter behavior can be seen in this code on GitHub. `</article>`

Andrés Almiray is a Java and Groovy developer and a Java Champion with more than 17 years of experience in software design and development. He has been involved in web and desktop application development since the early days of Java. He is a true believer in open source and has participated in popular projects such as Groovy, JMatter, Asciidoctor, and others. He is the founding member and current project lead of the Griffon framework and the specification lead for JSR 377.

learn more

- JDeferred library
- tutorial on futures, promises, and JDeferred
- Java 8 CompletableFuture (Javadoc)
- tutorial on Java 8 CompletableFuture

MERT ÇALIŞKAN

jsoup HTML Parsing Library

Easily parse HTML, extract specified elements, validate structure, and sanitize content.

Today, enterprise Java web application developers use HTML in every aspect of a project. This work is made difficult at times because parsing HTML content is a tedious task. Doing so without a parser framework is a most undesirable chore. Fortunately, there are a handful of Java-based HTML parsers publicly available. In this article, I will focus on one of my favorites, jsoup, which was first released as open source in January 2010. It has been under active development since then by Jonathan Hedley, and the code uses the liberal MIT license.

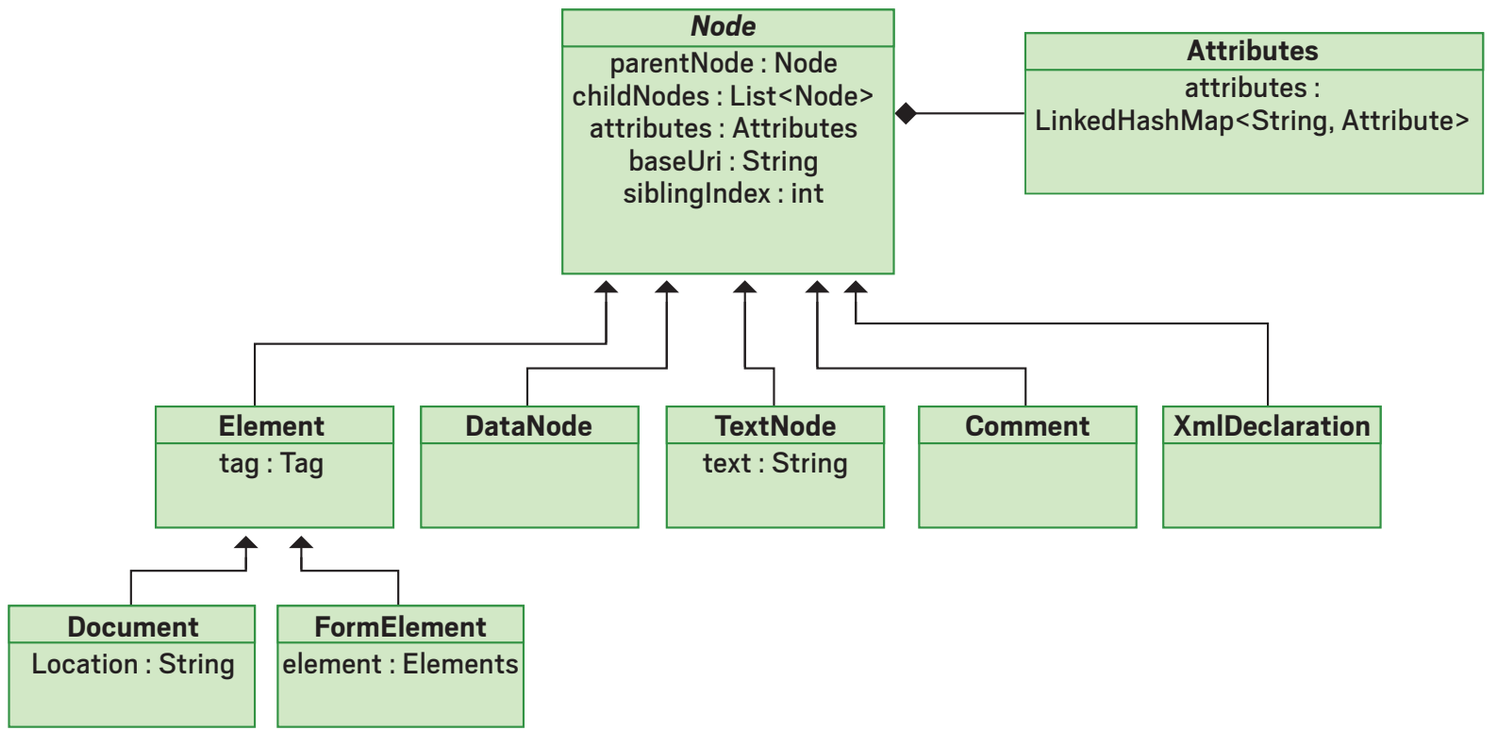


Figure 1. jsoup class diagram

What It Is

jsoup can parse HTML files, input streams, URLs, or even strings. It eases data extraction from HTML by offering Document Object Model (DOM) traversal methods and CSS and jQuery-like selectors.

jsoup can manipulate the content: the HTML element itself, its attributes, or its text. It updates older content based on HTML 4.x to HTML5 or XHTML by converting deprecated tags to new versions. It can also do cleanup based on whitelists, tidy HTML output, and complete unbalanced tags automagically. I will demonstrate these features with some working examples.

All the examples in this article are based on jsoup version 1.10.2, which is the latest available version at the time of this writing. The complete source code for this article is available on GitHub.

The DOM and jsoup Essentials

DOM is the language-independent representation of the HTML documents, which defines the structure and the styling of the document. Figure 1 shows the class diagram of jsoup framework classes. Later, I'll show you how they map to the DOM elements.

The `org.jsoup.nodes.Node` abstract class is the main element of jsoup. It represents a node in the DOM tree, which could either be the document itself, a text node, a comment, or an ele-

ment—that is, form elements—within the document. The `Node` class refers to its parent node and knows all the parent's child nodes.

The `Element` class represents an HTML element, which consists of a tag name, attributes, and child nodes. The `Attributes` class is a container for the attributes of the HTML elements and is composed within the `Node` class.

Getting Started

You can obtain the latest version of jsoup from Maven's Central Repository with the following dependency definition. The current release will run on any version of Java since Java 5.

```
<dependency>
    <groupId>org.jsoup</groupId>
    <artifactId>jsoup</artifactId>
    <version>1.10.2</version>
</dependency>
```

Gradle users can retrieve the artifact with

```
org.jsoup:jsoup:1.10.2
```

The main access point class, `org.jsoup.Jsoup`, is the principal way to use the functionality of jsoup. It provides base methods that can parse an HTML document passed to it as a file or an input stream, a string, or an HTML document provided through a URL. The example in **Listing 1** parses HTML text and outputs first the node name of the element and then the HTML text owned by the element, as shown immediately below the code.

Listing 1.

```
public class Example1Main {

    static String htmlText = "<!DOCTYPE html>" +
            "    <html>" +
            "    <head>" +
            "       <title>Java Magazine</title>" +
            "    </head>" +
            "    <body>" +
            "       <h1>Hello World!</h1>" +
            "    </body>" +
            "</html>";

    public static void main(String... args) {
        Document document = Jsoup.parse(htmlText);
        Elements allElements =
            document.getAllElements();
        for (Element element : allElements) {
            System.out.println(element.nodeName()
            + " " + element.ownText());
        }
    }
```

The output is

```
#document
html
head
title Java Magazine
body
h1 Hello World!
```

Ways to select DOM elements. jsoup provides several ways to iterate through the parsed HTML elements and find the requested ones. You can use either the DOM-specific `getElementBy*` methods or CSS and jQuery-like selectors. I will demon-

CSS and jQuery-like selectors are powerful compared with DOM-specific methods. They can be combined together to refine selection.

strate both approaches by parsing a web page and extracting all links that have HTML <a> tags. The code in **Listing 2** parses the Java Champions bio page and extracts the link names for all the Java Champions marked as "New!" (see **Figure 2**).

The marking was done by adding a <font> tag with text `New!` right next to the link. So, I will be checking for the content of the next-sibling element of each link.

Listing 2.

```
public class Example2Main {

    public static void main(String... args)
        throws IOException {
        Document document = Jsoup.connect(
            "https://java.net/website/" +
            "java-champions/bios.html" )
            .timeout(0).get();

        Elements allElements =
            document.getElementsByTag("a");
        for (Element element : allElements) {
            if ("New!".equals(
                 element.nextElementSibling()!=null
                 ? element.nextElementSibling()
                   .ownText()
                 : "")) {
                   System.out.println(
                       element.ownText());
```

Adam Bien
Xu Bin
David Blevins New!
Joshua Bloch
David Bock
Jonas Bonér
Bruno Bossola
Vincent Brabant*
Bill Burke*

C
Mert Caliskan New!
Michael Cannon-Brookes
Tasha Carl New!

Figure 2. Part of the HTML page to be parsed

```
            }
        }
    }
}
```

The same extraction of the links can also be done with selectors, as shown in **Listing 3**. This code extracts the links that start with `href` value `#`.

Listing 3.

```
public class Example3Main {

    public static void main(String... args)
            throws IOException {
        Document document = Jsoup.connect
                ("https://java.net" +
            " /website/java-champions/bios.html")
                .timeout(0).get();
        Elements allElements = document.select
                ("a[href*=#]");
        for (Element element : allElements) {
            if ("New!".equals(element
                    .nextElementSibling() != null
                    ? element.nextElementSibling
                    ().ownText() : "")) {
                System.out.println(element
                        .ownText());
            }
        }
    }
}
```

Selectors are powerful compared with DOM-specific methods. They can be combined together to refine selection. In the previous code examples, we are doing the `New!` text check by

ourselves, which is trivial. The example in **Listing 4** selects the `<font>` tag that contains the `New!` text, which resides after a link that has an `href` starting with the value `#`. This really shows the power of selectors.

Listing 4.

```
public class Example4Main {

    public static void main(String... args)
            throws IOException {
        Document document = Jsoup.connect
                ("https://java.net" +
            ".website/java-champions/bios.html")
            .timeout(0).get();
        Elements allElements = document.select
                ("a[href*=#] ~ font:containsOwn" +
                        "(New!)");
        for (Element element : allElements) {
            System.out.println(element
                    .previousElementSibling()
                    .ownText());
        }
    }
}
```

Here, the selectors locate the `<font>` tag as an element. I then call the `previousElementSibling()` method on it, so as to step one element back to the link. This `select()` method is available in the `Document`, `Element`, and `Elements` classes. Currently, jsoup does not support XPath queries on selectors. More information about selectors is available at the jsoup site.

Traversing nodes. jsoup provides the `org.jsoup.select.NodeVisitor` interface, which contains two methods: `head()` and `tail()`. By implementing an anonymous class from that interface and passing it as a parameter to the `document.traverse()` method, it is possible to have a callback when the node is first and last visited. The code in **Listing 5** uses this technique to traverse a simple HTML text and outputs all node details.

Listing 5.

```
public class Example5Main {

    static String htmlText = "<!DOCTYPE html>" +
            "<html>" +
            "<head>" +
            "<title>Java Magazine</title>" +
            "</head>" +
            "<body>" +
            "<h1>Hello World!</h1>" +
            "</body>" +
            "</html>";

    public static void main(String... args)
            throws IOException {
        Document document = Jsoup.parse(htmlText);

        document.traverse(new NodeVisitor() {
            public void head(Node node, int depth){
                System.out.println("Node start: "
                        + node.nodeName());
            }

            public void tail(Node node, int depth){
                System.out.println("Node end: " +
                        node.nodeName());
            }
        });
    }
}
```

The output from this traversal is as follows:

```
Node start: #document
Node start: #doctype
Node end: #doctype
Node start: html
Node start: head
Node start: title
Node start: #text
Node end: #text
Node end: title
Node end: head
Node start: body
Node start: h1
Node start: #text
Node end: #text
Node end: h1
Node end: body
Node end: html
Node end: #document
```

Parsing XML files. jsoup supports parsing of XML files with a built-in XML parser. The example in **Listing 6** parses an XML text and outputs it with appropriate formatting. Note once again how easily this is accomplished.

Listing 6.

```
public class Example6Main {

    static String xml =
         "<?xml version=\"1.0\"" +
         "encoding=\"UTF8\"><entries><entry>" +
         "<key>xxx</key>" +
         "<value>yyy</value></entry>" +
         "<entry><key>xxx</key>" +
         "<value>zzz</value>" +
         "</entry></entries></xml>";

    public static void main(String... args) {
        Document doc =
          Jsoup.parse(xml, "", Parser.xmlParser());
        System.out.println(doc.toString());
    }
}
```

As you would expect, the output from this is

```
<?xml version="1.0"encoding="UTF8">
<entries>
 <entry>
  <key>
   xxx
  </key>
  <value>
   yyy
  </value>
 </entry>
 <entry>
  <key>
   xxx
  </key>
  <value>
   zzz
  </value>
 </entry>
</entries>
```

It’s also possible to use selectors for picking up values from specified XML tags. The code snippet in **Listing 7** selects <value> tags that reside in <entry> tags.

Listing 7.

```
Document doc =
    Jsoup.parse(xml, "", Parser.xmlParser());
```

```
Elements elements = doc.select("entry value");
Iterator<Element> it = elements.iterator();
while (it.hasNext()) {
    Element element = it.next();
    System.out.println(element.nodeName() +
        " - " + element.ownText());
}
```

Preventing XSS attacks. Many sites prevent cross-site scripting (XSS) attacks by prohibiting the user from submitting HTML content or by enforcing the use of alternative markup syntax, such as markdown. A clever solution to prevent malicious HTML input is to use a WYSIWYG editor and filter the HTML output with jsoup's whitelist sanitizer. The whitelist sanitizer parses the HTML, and iterates through it and removes the unwanted tags, attributes, or values according to the whitelist built into the framework.

The example in **Listing 8** defines a test method that cleans up HTML text according to a simple text whitelist. This list, as you will see in a moment, allows only simple text formatting with HTML tags: `b`, `em`, `i`, `strong`, and `u`.

Listing 8.

```
@Test
public void simpleTextCleaningWorksOK() {
    String html = "<div>" +
        "<a href='http://www.oracle.com'>" +
        "<b>Hello + Reader</b>!</a></div>";
    String cleanHtml = Jsoup.clean(
        html, Whitelist.simpleText());
    assertThat(cleanHtml,
              is("<b>Hello Reader</b>!"));
    }
```

The `WhiteList` class offers prebuilt lists such as `simpleText()`, which limits HTML to the previous elements. There are other acceptance options, such as `none()`, `basic()`, `basicWithImages()`, and `relaxed()`.

Listing 9 shows an example of the usage of `basic()`, which allows these HTML tags: `a`, `b`, `blockquote`, `br`, `cite`, `code`, `dd`, `dl`, `dt`, `em`, `i`, `li`, `ol`, `p`, `pre`, `q`, `small`, `span`, `strike`, `strong`, `sub`, `sup`, `u`, `ul`.

Listing 9.

```
@Test
public void basicCleaningWorksOK() {
    String html = "<div><p><a " +
            "href='javascript:hackSystem()" +
            "'>Hello</a></div>";
    String cleanHtml = Jsoup.clean(html,
            Whitelist.basic());
    assertThat(cleanHtml, is("<p><a " +
            "rel=\"nofollow\">Hello</a></p>"));
}
```

As seen in the test, the script call is eliminated and the tags that are not allowed, such as `div`, are also removed. In addition, jsoup automatically completes unbalanced tags, such as the missing `</p>` in our example.

Conclusion

This article, which previously appeared in *Java Magazine* but has been updated here, shows only a subset of what jsoup can do. It also offers features such as tidying HTML, manipulating HTML tags' attributes or texts, and more. Put another way, any HTML processing you might need to do is a likely candidate for using jsoup. `</article>`

Mert Çalişkan is a Java Champion and coauthor of *PrimeFaces Cookbook* and *Beginning Spring* (Wiley Publications). He is the founder of AnkaraJUG, which is the most active Java user group in Turkey.

STEPHEN COLEBOURNE

Designing and Implementing a Library

The chief designer of Joda-Time lays out best practices for writing your own library.

There are many ways to build an application, but most of the time you will pull in a framework or two and a few libraries. Tooling tends to make this easy now, with build tools, such as Maven and Gradle, connecting to a central artifact repository of JAR files. Thanks to the world of open source, many thousands of libraries and frameworks are available to choose from (and most companies have an internal artifact repository with even more). But what makes a good library? How can it be designed well?

Styles of Library

When designing a library, it is useful to bear in mind some common styles that libraries fit into. Back in 2004, I identified two styles within Apache Commons: *broad and shallow* versus *narrow and deep.*

The broad and shallow style has many public methods for users to call (the broad part), each of which tends to do relatively little processing (shallow). Using the library focuses on finding the right class to call or create and then following the syntax and operations detailed in the Javadoc. Because the public methods are shallow, they tend to be fairly separate from the others, and it is often possible to split such a library into many smaller libraries. While often this style of library consists of classes with many static methods, they typically include instantiated classes, too. Examples of this style include Apache Commons Lang, Apache Commons IO, Google Guava, and Joda-Time.

The narrow and deep style has relatively few public methods for users to call (narrow), but each method tends to perform a decent amount of processing (deep). Using such a library tends to involve specific usage patterns that are documented at a high level—often outside the Javadoc. Examples of this style are XML parsers and templating libraries such as Apache FreeMarker. The key to making this approach work is to have an obvious, well-documented public API and to hide the internal classes.

In both styles, the library tends to have relatively small bounds. The result is that if you find the library you chose is buggy or not to your taste, it tends not to be too hard to replace it. This leads to a third style that might best be described as a "business" library. Here, the library is more specific, perhaps used primarily in an industry vertical, and adoption is a major architectural choice for an application. In my day job, I work on Strata, the Duke's Choice Award–winning library for finance, which is a classic open source example of this style. Most examples of this style are likely to be private and company-specific.

Dependencies

The ease of use and convenience of an artifact repository such as Maven Central makes it all too easy to just pull in dependencies. But when you do, consider how many other depen-

dencies that one dependency has. Too quickly, you find that your application has hundreds of dependencies, and you can face clashes between different versions, a situation termed *classpath hell*. As such, all good libraries strive to minimize their dependencies.

In my experience with Apache Commons and the Joda projects, I have found that broad and shallow libraries work best if they have no dependencies at all. Commons Lang, Commons IO, and Google Guava all have no dependencies.

There is an interesting case with the Joda-Time and Joda-Money libraries. Both of these broad and shallow libraries do have a dependency—Joda-Convert—but that dependency is *optional*. Most applications using Joda-Time do not need to have Joda-Convert on the classpath. Only if you use the additional features it provides will you include it.

In my experience, narrow and deep libraries tend to be more complex. As such, they often depend on a few other libraries, which is fine as long as the dependencies are limited. Larger business libraries typically have a larger set of dependencies, but this is usually fine because they are so important to the application that the library drives the dependencies of the application, not the other way around.

It doesn't make sense to depend on a library for a tiny amount of code, such as a few static utility methods. Instead, consider copying portions of the library into yours with a clear indication as to where the code came from. By keeping track of the copied code, it becomes easier to spot the point at which the additional dependency is worthwhile. Ideally, the code will be package-scoped when copied into your library, as it is not really part of your API.

I have found that broad and shallow libraries work best if they have no dependencies at all.

Finally, you should take extra care using Google Guava in a low-level library, because it tends to be widely used yet incompatible between releases, the classic classpath hell problem.

Integration

One tricky case can be *integration*, which is when a library needs to provide code to interoperate with other libraries. The most common way to do this is to release a core library and one additional library for each integration. With this approach, the core library is not burdened with the additional dependencies, but the user must pick the correct additional JAR file.

An interesting alternative is to use *optional dependencies*. With this approach, the library consists of a single JAR file with all the integration code included. However, each integration works only if the user also adds the integration JAR file to their classpath. This can be convenient for the user, as the integration can be made to work transparently.

Best practice normally favors the first approach, with separate JAR files. But when the integration code is relatively small and convenience is important, the second approach can be worthwhile to consider.

Structure

Most libraries consist of just a few packages, and libraries consisting of just one package are quite common. When designing a library, it can be useful to plan the package structure so it has a clear root package to aid first-time users. This is particularly important for narrow and deep libraries.

For example, the root package of the library `com.foo.shared` should contain the most important entry points to the library. Additional packages would contain classes of lower importance, say, `com.foo.shared.config` and `com.foo.shared.model`. Any code that should not be called directly by users should go in an internal package, such as `com.foo.shared.internal` or `com.foo.shared.impl`. In Java releases through Java SE 8, users can, of course, access these

internal packages. In Java SE 9 modules, however, it will be possible to properly restrict the internal packages so that users cannot access them directly at all.

In addition to modules, library designers should consider using package scope as much as possible. Package scope is hugely underused in Java generally, but it is a great tool for hiding your internal logic. Java SE 8, in particular, enables designers to make much greater use of package scope—thanks to the addition of methods on interfaces. Prior to Java SE 8, your library might have had an interface, a factory for creating instances, and an abstract class to allow for future change. Now, all three features can be combined: instances can be created using static methods on interfaces, and there is no need for an abstract class with default methods on interfaces. If the whole API can be defined by the interface, it is possible to make the implementation classes package-scoped, that is, created by the static factory methods on the interface. Suddenly, the public API has collapsed from maybe five public classes to one—a huge benefit for later maintenance.

Features and Growth

Many libraries start out from a simple need to share code between two projects. The code grows over time and eventually becomes unmanageable, whereas perhaps it should be split. The issue here is that the library grew without a mission statement. Why does this library exist? What problem is it solving? Why should you use this piece of shared code rather than writing it yourself?

By writing something down, often at the top of the home page of the project or in the README file, you set some boundaries for the library. When requests arrive for new features, it becomes easier to see whether the features are inside or outside the boundaries. This allows you to push back and reject the feature or perhaps create a new library.

If, however, the feature request is within the boundaries for the library, serious consideration should be given to including it. Libraries are shared code, and while perhaps your use cases didn't need the feature, someone else's might. But it is important to watch out for bloat, because as more features are added, it becomes harder for new users to learn the library and to find out what it contains. One way to judge whether inclusion warrants the added code is to consider how much code is being shared and what the nearest workaround is for callers. If the workaround is painful and the use case seems sound enough, the added code should probably go into the library.

> My experience is that if you follow a strict approach of never returning null, the **whole codebase becomes much clearer and safer for users.**

If you are fortunate enough to be writing a standalone library that isn't just a sharing of code between two applications, one point to bear in mind is that YAGNI ("you aren't gonna need it") typically does *not* apply. This is because your aim is to serve the needs of the niche that the library sits in so that users are confident that the code they might need will be there when they need it. Doing this may well require additional features or convenience methods beyond those of the minimal use case you have in mind.

Part of managing this growth over time is a plan for compatibility. In most cases, libraries should follow semantic versioning to clearly communicate the compatibility of each version. Tools are available to check this as part of the build process. To avoid classpath hell for your users, it is important to achieve binary compatibility, so that a new version of the library can be just dropped in. This can be painful for a library author, but when many others depend on your library

it is a necessity; a key part of the success of Joda-Time is that users can rely on the stable, compatible API, just as they can with the JDK.

I recommend the Apache License version 2.0 for most libraries. It is a good, well-written license that is widely used and easy for users to accept.

Good Design

A library generally sits at the bottom of an application. As such, it needs to be reliable and of high quality. When the application calls a library method, the application needs to be certain that the library will do what is asked of it. It turns out that the best way to arrange this is to use good, modern API design for the library.

Where possible, pass immutable objects into and out of the library. Immutable objects are far clearer for the user: they can be in only one state and will never be affected by any complex concurrency in the application. Of course, for the library designer, immutable objects entail more work. You need to write factory methods and, potentially, a mutable builder class. But the benefits pay off in the form of fewer bugs. (Consider this: if you allow users to pass mutable classes to your library, what happens when your users mutate them while your library is processing?)

The API should also be well defined with regard to `null`. The simplest approach is to reject `null` as an input to all methods. Java now has `Objects.requireNotNull()`, which can help here. An alternative is to accept `null` and treat it as a no-op or default value, but as I learned with Joda-Time, this approach is usually a very *bad* idea. As a general rule, methods that might have been defined to return `null` five years ago should now return `Optional`. My experience is that if you follow a strict approach of never returning `null`, the whole codebase becomes much clearer and safer for users.

Public API methods in the library should also follow sensible and consistent naming. It is vital for users to be able to find the functionality they are looking for using naming only. Consistency is key here. For example, it is fine to use an abbreviation if it makes sense in the context of the library, but use it consistently. And the general advice for initial capitalization of acronyms still applies: you should prefer `HttpResult` to `HTTPResult`.

To aid with compatibility, it is worth considering using a design for key API methods in which a request bean is passed in and a result bean is returned. This has the advantage that when a new feature is needed, you can simply add to the request/response bean rather than create a new method signature on the key API.

Documentation

Many developers find documentation to be an annoyance that gets in the way of writing the code. When building a library, you simply can't think like this. The end users of the library are typically not known to you—they can't just come ask you a question at your desk. Your only realistic option is to provide the documentation they need.

In practice, this means that the public API must have good and clear Javadocs. In addition, package-level Javadocs, overview Javadocs, and usage documents should be considered, particularly for narrow and deep libraries. These high-level documents should explain how to use the library and what the main entry point is, and they should identify which packages should not be used directly.

One absolutely vital piece of documentation is information about the thread safety of key lifecycle and session classes. For example, when you use an XML or JSON parser, there will typically be a single entry-point class. But should you create a new instance each time? Or should you store it in a static variable? The expected pattern, determined by the

thread safety of each class, must be documented. If you don't do this, your users might conclude that your library doesn't understand the importance and difficulty of concurrency.

A similar discussion applies to objects that hold external resources, such as streams and buffers. The documentation needs to be clear as to who should close the resource. If the library itself manages resources, for example, through an `ExecutorService`, it should implement `AutoClosable` and clearly document the usage pattern.

The final key piece of documentation is the license. While libraries within a company don't need this, open source libraries must have one. I recommend the Apache License version 2.0 for most libraries. It is a good, well-written license that is widely used and easy for users to accept.

Conclusion

To design a good library takes time, and it is a task that requires high-quality, clean code. After all, when building an application, all developers can tell whether they are using a good library or a bad one. So, if you are going to build a library, build it well. Your users will thank you. `</article>`

Stephen Colebourne (@jodastephen) is a Java Champion who has used Java since version 1.0. He is best known for his work on date and time, through Joda-Time and the Java 8 java.time.* packages. He has many other open source projects under the Joda and ThreeTen brands. Colebourne also writes blogs and speaks at conferences. He works at OpenGamma, producing software for the finance industry.

learn more

- Joda-Time library
- Example of Joda-Time's detailed Javadocs

FEATURED JDK ENHANCEMENT PROPOSAL

JEP 262: Built-in Support for TIFF Files

The wide range of image formats have not all enjoyed the same level of support in Java SE. The Image I/O Framework (javax.imageio), which is part of Java SE, provides a standard way to plug in image codecs. Codecs for some formats, such as PNG and JPEG, must be provided by all implementations. And other formats, such as BMP, have some built-in support. However, the widely used format TIFF is missing from the set of required codecs.

There have been multiple requests over the years for this format, from developers representing both small and large independent software vendors. The demand is even more relevant now because macOS uses TIFF as a standard platform image format.

JDK Enhancement Proposal (JEP) 262 proposes including a TIFF codec as part of javax.imageio. Suitable TIFF reader and writer plugins, written entirely in Java, were previously developed in the Java Advanced Imaging API Tools Project. This JEP proposes to merge this TIFF support into the JDK, alongside the existing image I/O plugins. The package will be renamed javax.imageio.plugins.tiff, and it will become a standard part of the Java SE specification. (The XML metadata format will be similarly renamed.)

As of the time of this writing, JEP 262 has been approved and finalized, and the TIFF support will appear in Java 9 when that release ships.

ORACLE CODE

Register Now

New One-Day, Free Event | 20 Cities Globally

Explore the Latest Developer Trends:

- DevOps, Containers, Microservices & APIs
- MySQL, NoSQL, Oracle & Open Source Databases
- Development Tools & Low Code Platforms
- Open Source Technologies
- Machine Learning, Chatbots & AI

Live for the Code

Find an event near you:

developer.oracle.com/code

ORACLE®

PER MINBORG

Database Actions Using Java 8 Stream Syntax Instead of SQL

Speedment 3.0 enables Java developers to stay in Java when writing database applications.

Why should you need to use SQL when the same semantics can be derived directly from Java 8 streams? If you take a closer look at this objective, it turns out there is a remarkable resemblance between the verbs of Java 8 streams and SQL commands, as summarized in **Table 1**.

Streams and SQL queries have similar syntax in part because both are declarative constructs, meaning they describe a result rather than state instructions on how to compute the result. Just as a SQL query describes a result set rather than the operations needed to compute the result, a Java stream describes the result of a sequence of abstract functions without dictating the properties of the actual computation.

The open source project Speedment capitalizes on this similarity to enable you to perform database actions using Java 8 stream syntax instead of SQL. It is available on GitHub under the business-friendly Apache 2 license for open source databases. (A license fee is required for commercial databases.) Feel free to clone the entire project.

About Speedment

Speedment allows you to write pure Java code for entire database applications. It uses lazy evaluation of streams, meaning that only a minimum set of data is actually pulled from the database into your application and only as the elements are needed.

In the following example, the objective is to print out all `Film` entities having a rating of PG-13 (meaning "parents are strongly cautioned" in the US). The films are located in a database table represented by a Speedment `Manager` variable

SQL COMMAND	JAVA 8 STREAM OPERATIONS
FROM	`stream()`
SELECT	`map()`
WHERE	`filter()` (BEFORE COLLECTING)
HAVING	`filter()` (AFTER COLLECTING)
JOIN	`flatMap()` OR `map()`
DISTINCT	`distinct()`
UNION	`concat(s0, s1).distinct()`
ORDER BY	`sorted()`
OFFSET	`skip()`
LIMIT	`limit()`
GROUP BY	`collect(groupingBy())`
COUNT	`count()`

Table 1. SQL commands and their counterpart verbs in Java 8 streams

named `films` in this code snippet:

```
long count = films.stream()
    .filter(Film.RATING.equal("PG-13"))
    .count();

System.out.format(
    "There are %d films rated 'PG-13'", count
);
```

Now to the best part: immediately before the stream is about to start, Speedment will introspect the stream pipeline and determine that an equivalent but more efficient stream can replace the present stream.

In addition, by enabling logging, you can see the exact rendering of the SQL query, as follows:

```
SELECT COUNT(*) FROM 'sakila'.'film' WHERE
    ('rating' = 'PG-13' COLLATE utf8_bin)
```

So, in fact, the stream does not pull in a single film. It merely lets the database count the given films, and then it returns the result directly to the Java application. The code above will produce the following output:

```
There are 223 films rated 'PG-13'
```

As you can see, there is no SQL code in the Java application. (Note: In the examples throughout this article, I used a MySQL database. If another database type, such as Oracle Database or PostgreSQL, were used, Speedment would render the stream to another SQL query variant depending on that database's specific syntax capability. This is something Speedment handles automatically.)

Speedment contains a tool that connects to an existing database, extracts the schema metadata, and generates Java code.

How It Works

If you have worked with Java 8 streams before, the previous example and those that follow will look familiar. Because a Java 8 `Stream` is an interface, there can be many different implementations of streams. Speedment comes with its own `Stream` implementations that connect seamlessly to database tables and lazily pull in rows as the streams are being consumed. Furthermore, a Speedment stream introspects its own pipeline before it starts. This means stream operations such as `filter()` and `sorted()` can be moved from the stream's pipeline into a SQL query (more specifically, into its WHERE and ORDER BY clauses, respectively), effectively eliminating the need for writing SQL code altogether. Let's take a closer look at how this affects Java database applications.

Basic Examples

Throughout this article's examples, I used a sample film database called sakila. It was developed by the MySQL team and has been available for about 10 years. The sample database is open source and available directly from Oracle.

Speedment contains a tool that connects to an existing database, extracts the schema metadata, and generates Java code. The code generator generates objects corresponding to the selected database tables. Consequently, a Java entity interface `Film` is generated from the `film` table with getters and setters for all its columns. For each table, a corresponding `Manager` is also generated and, thus, a `FilmManager` is generated for the `film` table as well. `Managers` are used for creating streams and other table-related operations.

Creating a Stream

Suppose you want to write an application that prints out all `Film` entities. With Speedment, it looks like this:

```
films.stream()
    .forEach(System.out::println);
```

On the first line, a Speedment stream is created using the variable `films` (of type `FilmManager`). You will see later in the article how `Manager` variables can be obtained. On the second line, the stream is consumed by sending all its elements to `System.out`. The partial output of the code snippet (only some columns and only the first three films) is shown below:

```
filmId = 1, title = ACADEMY DINOSAUR, rating = PG
filmId = 2, title = ACE GOLDFINGER, rating = G
filmId = 3, title = ADAPTATION HOLES, rating = NC-17
```

There are 1,000 films present in the sample database, and the table has 13 columns.

Understanding Optimization

Optimization of stream operations during introspection is not limited only to `filter()` operations. Consider the following stream example:

```
long count = films.stream()
    .filter(Film.RATING.equal("PG-13"))
    .filter(Film.LENGTH.greaterThan(75))
    .map(Film::getTitle)
    .sorted()
    .count();

System.out.format("Found %d films", count);
```

This code creates a stream in which only those films having a rating of PG-13 and a length of more than 75 minutes appear. After filtering, the `Film` elements are mapped to String elements. This is possible because each film's title is extracted using the `Film::getTitle` method reference. After this step, the stream of string elements is sorted (according to the previously extracted titles, in natural order), and finally all the sorted strings are counted.

Upon introspecting the stream, Speedment determines neither the `map()` nor the `sorted()` operator has any effect on the final `count()` outcome, because these operators do not change the number of elements in the stream. The operator `map()` changes only the *type* of the elements, and `sorted()` changes only the *order* in which the elements appear in the stream. Therefore, eventually the stream is reduced and rendered to the following SQL query:

> With Speedment, as with any object-relational mapping, **entities can be created, updated, and deleted.**

```
SELECT COUNT(*) FROM 'sakila'.'film' WHERE
('rating' = 'PG-13' COLLATE utf8_bin) AND ('length'>75)
```

The code above produces the following output:

```
Found 181 films
```

So that Speedment can optimize a given stream, use predicates derived from fields instead of anonymous lambdas. That is, do this:

```
filter(Film.LENGTH.greaterThan(75))
```

instead of doing this:

```
filter(f -> f.getLength() > 75)
```

Classifying Films

Now suppose you want to group all the films by `rating` and produce a Java map with all these groups. This can be done

using standard Java 8 semantics applied to a Speedment stream:

```
Map<String, List<Film>> map = films.stream()
   .collect(
      Collectors.groupingBy(
         // Apply this classifier
         f -> f.getRating().orElse("none")
      )
   );

map.forEach((k, v) ->
   System.out.format(
      "Rating %-5s maps to %d films %n",
      k, v.size()
   )
);
```

Because the `rating` column is defined as NULLABLE in the sample database, Speedment generates a corresponding `getRating()` method that returns an `Optional<String>` rather than just a String. This helps avoid accidental null pointer exceptions in the application. Thus, if a film is not rated (its rating is NULL in the database), the `getRating()` method returns an `Optional.empty()` and the classifier defaults to none.

The previous code might produce the following output:

```
Rating PG-13 maps to 223 films
Rating R     maps to 195 films
Rating NC-17 maps to 210 films
Rating G     maps to 178 films
Rating PG    maps to 194 films
```

This is consistent with the earlier example in which there were 223 films rated PG-13.

One-to-Many Relations

In the example database, the tables `film` and `language` have a relation via a foreign key from `film.language_id` to `language.language.id`. If the task were to print all films that are in English, the following example is a way of doing it:

```
languages.stream()
   .filter(Language.NAME.equal("English"))
   .flatMap(films.finderBackwardsBy(
        Film.LANGUAGE_ID))
   .forEach(System.out::println);
```

This is how it works: first, the English language is filtered out. Then the actual relation between the tables is specified by applying a `flatMap()` operator. This operator takes a `Function` that maps from a `Language` to a `Stream<Film>` whereby the latter contains only films matching the particular language. In short, the penultimate line takes you from a `Stream<Language>` to a `Stream<Film>`. This is a relation commonly referred to as a *one-to-many relation* because many films can point to the same language. Speedment is aware of the columns having foreign keys and only those columns can be passed to the `finderBackwardsBy()` method, which ensures full relational integrity at compile time.

CRUD Operations with Streams

With Speedment, as with any object-relational mapping (ORM), entities can be created, updated, and deleted. These operations can be integrated with Java 8 streams, as illustrated in the example below. Here, all language entities with a name "Deutsch" are to be renamed "German":

```
languages.stream()
    .filter(Language.NAME.equal("Deutsch"))
    .map(Language.NAME.setTo("German"))
    .forEach(languages.updater());
```

First, the language to be changed is filtered out. Then a `map()` operation is added to the stream that, when applied, takes a `Language` entity and returns a (possibly new) `Language` entity with the "name" field changed to "German." On the last row, the language `updater()` is called for each language in the stream. As its name implies, a language `updater()` takes a `Language` and then locates and updates the affected row in the database.

Writing web applications and REST endpoints using, for example, Spring Boot or Java EE is straightforward.

Remember, Speedment optimizes the stream and pulls in only those elements fitting the filter.

Get Started with Speedment

Speedment is available in the Maven Central Repository. Enable Speedment in your projects by adding the following coordinates in your pom.xml file:

```
<build>
  <plugins>
    <plugin>
      <groupId>com.speedment</groupId>
      <artifactId>speedment-maven-plugin</artifactId>
      <version>${speedment.version}</version>
    </plugin>
  </plugins>
</build>

<dependencies>
  <dependency>
    <groupId>com.speedment</groupId>
    <artifactId>runtime</artifactId>
    <version>${speedment.version}</version>
    <type>pom</type>
  </dependency>
</dependencies>
```

The Speedment Maven plugin adds code generation to the project. It also includes three other targets enabling you to automate your Maven builds.

Make sure you use the latest release available and add the runtime JDBC dependency for your selected database type under the `<dependencies>` tag (as you would in any database application). You must run your application under JDK 8u40 or later. Speedment is a completely self-contained runtime with no external transitive dependencies. This is important because it allows you to avoid potential version conflicts with other libraries and the ever-lurking "JAR hell." Furthermore, there is a "deploy" variant available where all Speedment runtime modules have been packed together into a single compound JAR file.

Initializing Speedment

Upon code generation, entities and managers are generated for each table. At the same time, an `Application` and an `ApplicationBuilder` are generated. These classes can be used to manage the lifecycle of Speedment in your application. Here is a typical example of how to create, use, and stop a Speedment application:

```
SakilaApplication app =
  new SakilaApplicationBuilder()
    .withPassword("sakila-password")
    .withLogging(LogType.STREAM)
    .build();

LanguageManager languages =
    app.getOrThrow(LanguageManager.class);
```

```
languages.stream()
    .forEachOrdered(System.out::println);

app.stop();
```

`SakilaApplicationBuilder` is a configuration class that can be used to set a number of configuration parameters. In the example above, the password that is going to be used when connecting to the database is set. Logging of all streams is enabled, causing stream SQL rendering to be shown in the logs. Once the `build()` method is called, the configuration is frozen (that is, an immutable configuration object is created) and the Speedment application is started and is ready to use. After the application is built, a `LanguageManager` is obtained from the application. This manager can be used to create streams from the language table. After that is done, a stream of all languages is created and the entities are printed out. Lastly, the Speedment application is stopped and any resources being held (such as database connections in a connection pool) are released.

Create the `Application` instance only once in your application and keep it running until your application exits. Pass the `Application` instance to your business logic or inject it in your classes using Spring Boot or Java EE, for example.

Integration with Spring Boot

It is easy to integrate Speedment with Spring Boot. Here is an example of a Speedment configuration file for Spring:

```
@Configuration
public class AppConfig {
    private @Value("${dbms.username}") String username;
    private @Value("${dbms.password}") String password;
    private @Value("${dbms.schema}") String schema;

    @Bean
    public SakilaApplication getSakilaApplication() {
        return new SakilaApplicationBuilder()
            .withUsername(username)
            .withPassword(password)
            .withSchema(schema)
            .build();
    }

    // Individual managers
    @Bean
    public FilmManager getFilmManager(
         SakilaApplication app
    ) {
        return app.getOrThrow(FilmManager.class);
    }
}
```

Therefore, when you need to use a `Manager` in a Spring model-view-controller, you can now simply auto-wire it:

```
private @Autowired FilmManager films;
```

Serving Up a REST Endpoint

Writing web applications and REST endpoints using, for example, Spring Boot or Java EE is straightforward. In the following example, the task is to write a method `serveFilms(String rating, int page)` that returns a stream of `Film` entities. The `rating` controls the stream, allowing only films with the given rating to appear in the stream. If rating is `null`, all films are returned. Furthermore, the

Open source Speedment makes life easier for Java developers and allows them to express database queries in pure Java, using well-known APIs.

page parameter indicates which page to render on the web user's screen. The first page is page 0, the next is 1, and so on. Finally, all films are ordered by length. All this can be done with the following code:

```
private static final int PAGE_SIZE = 50;

private Stream<Film> serveFilms(
    String rating, int page)
{

  Stream<Film> stream = films.stream();

  if (rating != null) {
    stream =
      stream.filter(Film.RATING.equal(rating));
  }

  return stream
    .sorted(Film.LENGTH.comparator())
    .skip(page * PAGE_SIZE)
    .limit(PAGE_SIZE);
}
```

This code snippet could easily be improved to take parameters that specify a dynamic sort order and a custom page size.

Performance and Future Work

Under the hood, Speedment converts a ResultSet to a Stream. The raw conversion overhead compared to reading the ResultSet with custom JDBC code and then converting each row to an entity is in every practical aspect negligible.

Speedment further supports parallel streams so you can process the results from a database query *in parallel* and divide the work using the CommonForkJoinPool or any other thread pool of your choice.

The Speedment runtime can be deployed under Java 9, and then it supports the improved Java 9 Stream API including the new methods Stream::takeWhile and Stream::dropWhile.

Currently, Speedment supports lazy joining of tables. Semantic joins, by which it is possible to eagerly join tables, are being planned. This capability will make it easier to express different kinds of joins and will improve performance for larger joins.

Speedment is open source for open source databases and currently supports MySQL, PostgreSQL, and MariaDB. A commercial implementation, Speedment Enterprise, supports Oracle Database, Microsoft SQL Server, and IBM DB2 and DB2/400. Support for additional database types is in the works.

Conclusion

Open source Speedment makes life easier for Java developers and allows them to express database queries in pure Java while using well-known APIs (such as java.util.Stream) that are interoperable with a large number of other libraries. Using introspection, Speedment is able to render a Java 8 stream pipeline to SQL and lazily pull in relevant elements only as the application needs them.

Per Minborg (@PMinborg) is a Palo Alto, California–based inventor, developer, JavaOne alumnus, and coauthor of the publication *Modern Java*. He has 20 years of Java coding experience and runs Minborg's Java Pot blog. Minborg is a frequent contributor to open source projects.

learn more

- Code for the examples in this article
- Speedment Javadoc

DEVOXX™ POLAND

21 - 23 June 2017, Krakow

3 DAYS

100 SPEAKERS

2500 DEVELOPERS

250 EURO

REGISTER NOW | WWW.DEVOXX.PL

SIMON ROBERTS

Quiz Yourself

Intermediate and advanced test questions

In the hope that you liked the mix of questions in my previous column, I'll continue with questions that simulate the level of difficulty of the Oracle Certified Associate exam, which contains questions for a more preliminary level of certification. I also include some more-advanced questions that simulate those from the 1Z0-809 Programmer II exam, which is the certification test for developers who have been certified at a basic level of Java 8 programming knowledge and now are looking to demonstrate greater expertise.

As before, I avoid beginner questions and stay at the intermediate and advanced levels, which are marked as such.

Question 1 (intermediate). Given this fragment:

```
String[][] x = new String[1][]; // line n1
x[0][0] = "Fred"; // line n2
System.out.println("name is " + x[0][0]);
```

What is the result?

a. Compilation fails at line n1.
b. Compilation fails at line n2.
c. An exception is thrown at line n2.
d. `name` is `Fred`.
e. `name` is null.

Question 2 (intermediate). Given this code:

```
public void aMethod() {
    // line n1
    for (int x = 0; x < 10; x++) {
        // line n2
    }
    // line n3
}
```

Which three of the following are true?

a. Inserting `{ int x = 100; }` at line n1 results in a compilation error.
b. Inserting `int x = 100;` at line n1 results in a compilation error.
c. Inserting `{ int x = 100; }` at line n2 results in a compilation error.
d. Inserting `int x = 100;` at line n2 results in a compilation error.
e. Inserting `int x = 100;` at line n3 results in a compilation error.

Question 3 (advanced). Given a directory hierarchy such that the root directory / contains a subdirectory a/, that subdirectory a/ contains a subdirectory x/, and that subdirectory x/ contains a subdirectory y/, and also given that a file `a.txt` is in subdirectory x/ and a file `b.txt` is in subdirectory y/, like this:

```
/
└─ a/
   └─ x/
      ├─ a.txt
      └─ y/
         └─ b.txt
```

Suppose that all the files and directories are plain and regular in nature and fully accessible by the executing code, that the following setup code runs with a current working directory of a/, and before any other code:

```
Path dir = Paths.get("./x").toAbsolutePath();
```

Which of the following fragments produce the following output?
(All compile and run normally.)
/a/x/a.txt

a.
```
Files
        .list(dir)
        .forEach(System.out::println);
```
b.
```
Files
        .walk(dir)
        .filter(Files::isDirectory)
        .forEach(System.out::println);
```
c.
```
Files
        .walk(dir.normalize())
        .filter(Files::isRegularFile)
        .forEach(System.out::println);
```
d.
```
Files
        .find(dir, 1, (x, y) -> Files.isDirectory(x))
        .forEach(System.out::println);
```
e.
```
Files
        .find(dir.normalize(), 1, (x, y) ->
            !Files.isDirectory(x))
        .forEach(System.out::println);
```

Question 4 (advanced). Given these descriptions:

1. Access by a single-threaded program
2. Access by a multithreaded program
3. High frequency of reading
4. High frequency of writing
5. Low frequency of reading
6. Low frequency of writing

Which combination would make it appropriate to use a `CopyOnWriteArrayList`?

a. 1 and 4 in the same program
b. 1 and 6 in the same program
c. 2 and 4 in the same program
d. 2, 3, and 6 in the same program
e. 2, 4, and 5 in the same program

Question 1. The correct answer is option C. This question probes the nature of Java's multidimensional arrays. In fact, it's often said that Java does not have multidimensional arrays, but that it allows arrays of anything, including arrays of arrays. While this position is debatable (the language specification both asserts it and contradicts it) and it is certainly a fine distinction, it embodies an important and useful truth. Therefore, in the declaration of `x` in line n1, `x` is not an array, and even more, `x` is not a two-dimensional array. Rather, `x` is, as always, a reference. The variable `x` can refer to a simple, single-dimensional array, but the elements of that array must themselves be arrays (or be null). Those secondary arrays must in turn be arrays of references to String, or they must be null.

So far, so good. What does `x` refer to in this case? The initialization expression `String[1][]` is interesting. It instantiates a single array with one element. The element type of that array is "array of Strings," but because the second square-

bracket pair is empty, no secondary array is created. Notice that this syntax is legal (and useful), so the compilation failure proposed in option A is false.

Whenever Java allocates heap memory for an object (and arrays are objects), the memory is zeroed before any further initialization (such as invoking a constructor) occurs. This means that the single element of the array that is created contains a null pointer. That single array element is `x[0]`, and given that no other assignment is made to it in the code, its value is null. The code `x[0][0]` is syntactically legitimate, so option B is false. It would be interpreted as follows: "Follow the reference in the variable `x` to an array. Take the first element of that array, and follow that reference to another array. Take the first element of that array and use it as a reference to a String." Of course, in this case, `x[0]` is a null pointer, so the attempt to find the subarray throws a `NullPointerException`, which means that option C is true.

Options D and E are both false, because the code never prints anything; it crashes with the `NullPointerException` before that point. In fact, if line n2 did not exist, the same `NullPointerException` would occur at the output line, because the print expression also attempts to dereference the null pointer that is `x[0]`.

Question 2. The correct answers are options B, C, and D. In Java, variables are *block scoped*. Generally, that means that a variable is visible from the point of its declaration to the end of the block that encloses the declaration. In this case, that block is the following:

```
{
    // general code, x not in scope because
    // it's not yet declared
    int x = 99;
    // general code, x in scope
} // scope of x ends here
```

On this basis, option A does not cause a compilation error, because the declaration of `int x` that it contains is entirely local to the block. Hence, option A is incorrect.

However, in option B, the variable introduced has a scope that extends throughout the `for` loop, the block associated with that loop, and all the way to the closing curly brace following line n3. As a result, the variable declared in the `for` loop becomes a duplicate variable `x` and the code would not compile. Because of this, option B is a correct answer.

A variation on the simple description of scope above applies to `for` loops, formal parameters of methods, try-with-resources, and `catch` blocks. These structures have broadly similar forms with variable declarations enclosed in parentheses and with a block immediately following the closing parenthesis. In these situations, the scope of the variable begins with its declaration, but the scope ends with the closing brace of the following block. If a `for` loop has a single subordinate statement, rather than a block, the scope ends at the end of that statement. It's probably a very bad idea stylistically to leave out the braces, even when only a single statement is controlled by the loop. Therefore, the preferred style is the following:

```
for (int x = 0; x < 10; x++) {
    // x in scope
} // scope of x ends here
```

In particular, notice that although a variable declaration does not escape the block that contains its scope, it does penetrate inside any nested blocks. In this case, any attempt to define a new variable `x` inside the `for` loop (whether surrounded by a block of its own or not) will fail, because the `x` declared in the `for` loop's control structure results in the new declaration being a duplicate. Because of this, options C and D both result in compilation errors and they are, therefore, correct answers.

Because the declaration of `int x` in the `for` loop has a scope that ends with the end of the block that is subordinate to that loop, there's no variable `x` in scope at line 3. As a result, adding the declaration in option E does not cause any problems, and option E is incorrect.

A side note on exam questions: as a rule, questions try to avoid using negatives, because they're easy to miss. In this case, notice that the question asks a positive question, but the options refer to "result in a compilation error." This might be unexpected, but be sure to read the question that's actually in front of you and try to avoid letting your brain make assumptions. Programmers know that close attention to detail is critical in this line of work, so be sure to use that skill when answering questions, too.

Question 3. The correct answer is option E. This is a question that demands a certain knowledge of Java's APIs. There aren't many questions of this kind, because there's an argument that this kind of information can readily be looked up and need not be learned. On the other hand, it's not a bad idea to have a broad knowledge of the kinds of features available in the APIs, because it's common to see handwritten code that duplicates (and commonly does so with errors) capabilities that are provided in a core API. After all, if you don't even know the capability exists, you're not very likely to look up the details of how to use it. In a learning situation, such as reading this article, it's often interesting to discover what features are available that might be unfamiliar.

In this question, you're told that all the code compiles and runs, so from that you know that there must be five static methods in a class called `Files`. By the way, this class full of utilities was introduced with Java 7, so it's actually new enough that many programmers haven't found it yet. `Files` offers many useful methods for file manipulation, reading, and writing, and if this class is new to you, it's worth a look if you ever have to manipulate files. The methods used in this question are `list`, `walk`, `find`, `isDirectory`, and `isRegularFile`.

The methods `isDirectory` and `isRegularFile` behave as their names suggest. They take a `Path` object as an argument and return a Boolean value indicating whether `Path` describes a directory or a regular file (that is, a file that can hold data). They both actually have a second argument that indicates how to handle links. The methods use varargs, so the second arguments are optional—which is why it doesn't show up in these examples.

The method `Files.list` creates a stream of `Path` objects that enumerate the contents of the argument directory. The `Path` class, as can reasonably be inferred from the given source code, represents a file or directory name, possibly including path information. It's also reasonable to infer that the `toString` method of a `Path` returns a reasonable textual representation. If this weren't the case, none of the options could create the output required. However, the `Files.list` method enumerates the entire contents of the directory that it examines, which means that in this case, it refers not only to the `a.txt` file but also to the `y` directory. For this reason, option A is incorrect.

Another point about the `Path` class is that it can represent either relative or absolute paths—for example, `./x/a.txt` or `/a/x/a.txt`, respectively. In this case, the preamble code forces the `Path` object into an absolute-path mode, but the `Path` referred to by `dir` is actually `/a/./x/`, and the dot stays in the output. This, too, means that option A must be incorrect. To remove this excess dot, you can invoke the `normalize` method on the `Path` object. This results in a `Path` that has had references to `.` and `..` cleaned up without changing the target of the `Path` object. This fact allows you to reject options B and D for the same reasons.

Next, consider the `Files.walk` method. This method creates a `Stream<Path>` that enumerates all the items in the starting directory and subdirectories. However, because this

descends into subdirectories, the stream created in option B would initially include directories `x` and `y` and files `a.txt` and `b.txt`. A filter is applied to this stream that will allow only directories to pass, and this means that the output will be `/a/./x` and `/a/./x/y`. This means that the wrong items are shown, and the formatting has an excess dot. Therefore, option B is incorrect.

Option C also uses the `walk` method. It starts by calling `normalize` on the starting directory, so the format will be correct, and it filters out the directories, leaving the files. However, the output includes all the files in the tree and, therefore, will include both `/a/x/a.txt` and `/a/x/y/b.txt`. Because of this, option C is incorrect.

The third method that you must consider is the `Files.find` method. This is very similar to the `walk` method, in that it creates a `Stream<Path>` that represents items pulled recursively from the directory hierarchy. The difference is that the `find` method can exclude items from that stream. To be fair, you can remove items using a filter applied to the stream obtained from a walk operation. That's illustrated in options B and C. However, downstream filtering is typically less efficient. In the case of a `find` operation, the path and file attributes are passed into the third argument of the `find` method (which is `BiPredicate<Path, BasicFileAttributes>`). These file attributes are read when the directory is first scanned. In contrast, the downstream filter—as in options B and C—requires that the information be read a second time, which is less efficient.

The `BiPredicate` operation must return `true` if a contender `Path` is to be included in the stream that `find` creates. On that basis, option D would enumerate the directories, not the files, and must, therefore, be incorrect.

The `find` method also has the ability to limit the depth of recursion down the directory tree. This is the purpose of the second argument (the numeric one). In this case, the value `1` allows examination of the contents of the directory that is specified in the first argument. The value `1` in option E is sufficient to prevent the stream from including the file `b.txt`. Also, because the first argument is `dir.normalize()`, the format of the output is correct and does not include the undesired dot. Therefore, option E is correct.

As a side note, the `find` method takes an optional fourth argument that allows you to specify whether the recursion should follow links or not.

Question 4. The correct answer is option D. The `CopyOnWriteArrayList` class is defined in the `java.util.concurrent` package. Functionally, it provides an implementation of the `List` interface, but it is specifically designed to help you handle scalability issues.

If a system is "scalable," this means that as you add more compute hardware to it, it becomes capable of handling more work in the same amount of time. Ideally, if you doubled the amount of hardware, you'd double the throughput. However, usually you get diminishing returns. How badly those throughput returns diminish is defined mathematically by Amdahl's law. In simplified terms, Amdahl's law says that the more often, or the longer, that threads have to wait for one another, the less the system is able to benefit from adding more hardware to it—that is, the less scalable it is. Modern systems are commonly expected to scale well, so it's important to design them in a way that minimizes the time that threads have to wait for each other.

The copy-on-write structures in Java's concurrent API address a very specific situation. If a program has a data structure that is being accessed at very high rates by multiple threads, but all of those threads are reading and never altering the data, no locking is needed, and the threads need not wait for one another. However, if any thread wants to make a change, normally no other threads can be allowed to access the data while that change is being made, and a great deal of waiting results. That waiting causes a loss of scalability.

Now suppose that a program does a lot of concurrent reading, but occasionally a thread wants to modify the data. One approach would be to have the read operations be unprotected (so no loss of scalability occurs). This means that no thread can ever be permitted to modify the data. Therefore, when a modification must be made, the thread that wants to do this starts by making a copy of the data—that's a read operation, so it's completely safe. Then, in the private copy, the writing thread can safely make an update. The reading threads can continue while this is going on, although they are getting "stale" data at this point. If that staleness matters (it often doesn't), this approach is unsuitable. At the point that the change has been completed, the structure can start directing reading threads at the updated data set.

Notice that this copy operation could be hugely expensive for a large list, and on that basis, this approach is useful only if all of the following are true:

- Many threads need concurrent read access.
- It's very rare for threads to modify the data.
- You need to maintain the scalability of the system.
- It's OK that reading threads are seeing data that's a little stale from time to time.

A single-threaded system is not scalable anyway, because it has no ability to use additional CPUs. Therefore, item 1 must be invalid and item 2 is a requirement. Because high read rates and low write rates are needed, you can see that items 3 and 6 are also requirements, which means that option D is the only correct option. `</article>`

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who teaches at many large companies in Silicon Valley and around the world. He remains involved with Oracle's Java certification projects.

THE BANGLADESH JUG

The Bangladesh JUG (JUGBD) started in August 2013 as an effort to bring the Java developers of Bangladesh together. Java is one of the most popular languages in Bangladesh. Despite that, there has been a lack of an active and organized community to bring developers under the same roof to discuss all things Java. JUGBD is an attempt to address these issues, particularly over great tea.

The history of software development in Bangladesh goes back roughly 20 years, and software export began around the time the internet started getting popular. Consequently, Java was one of the first languages to enjoy widespread adoption among Bangladeshi developers. A tight-knit community grew around it but eventually became inactive. JUGBD was formed to revive the community and ensure that it is self-sustaining.

JUGBD organizes physical meetups sponsored by local software firms, as well as occasional unsponsored virtual meetups. These meetups primarily consist of a series of talks given by Bangladeshi as well non-Bangladeshi developers. Topics range from absolute beginner level to make students interested in Java, to advanced level aimed at seasoned developers. The next meetup is expected to attract as many as 100 professional developers and students. Additionally, individual community members participate in the Java Community Process, and the JUGBD organization aims to participate in the process in the near future.

You can visit JUGBD at its website, its Facebook group, or the meetup group.

//contact us /

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@omeda.com, who will do whatever they can to help.

Where?

Comments and article proposals should be sent to our editor, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A-3133, Redwood Shores, CA 94065, USA.

- Subscription application
- Download area for code and other items
- *Java Magazine* in Japanese